

AD-A189 554

VALIDATING AND EVALUATING ADA'S (TRADE MARK)
REPRESENTATION CLAUSES AND I. (U) AIR FORCE INST OF
TECH WRIGHT-PATTERSON AFB OH SCHOOL OF ENGI

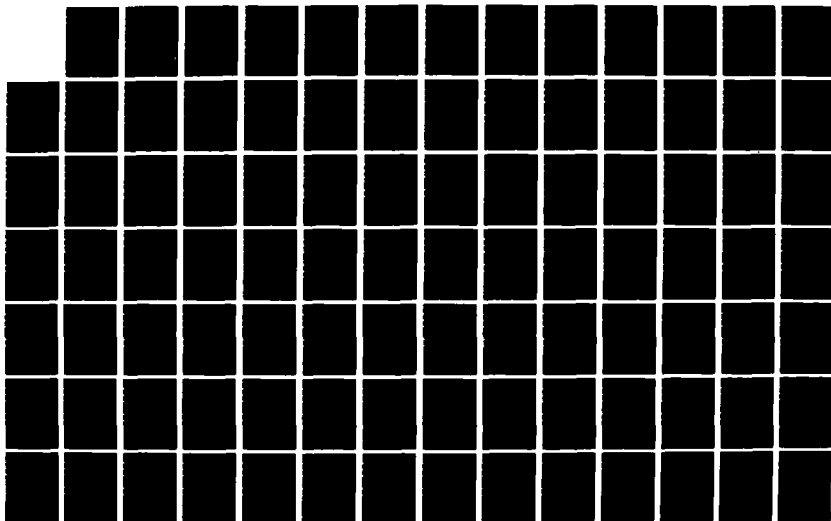
1/2

UNCLASSIFIED

D O JOYCE DEC 87 AFIT/GCS/MA/87D-2

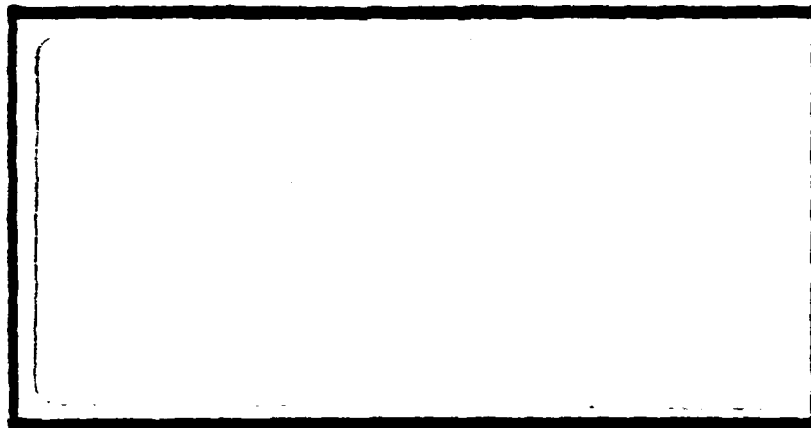
F/G 12/5

NL



DTIC FILE COPY

AD-A189 554



DTIC
ELECTE
MAR 03 1988
S E D

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY
AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

This document has been approved
for public release and sale in
distribution is unlimited.

88 3 01 118

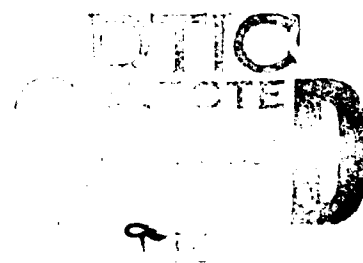
AFIT/GCS/MA/87D-2

VALIDATING AND EVALUATING ADA'S*
REPRESENTATION CLAUSES AND
IMPLEMENTATION-DEPENDENT FEATURES
ON MIL-STD-1750A ARCHITECTURE

THESIS

Daniel O. Joyce, Captain, USAF

AFIT/GCS/MA/87D-2



* Ada is a registered trademark of the U.S. Government (AJPO).

Approved for public release; distribution unlimited.

AFIT/GCS/MA/87D-2

VALIDATING AND EVALUATING ADA'S* REPRESENTATION CLAUSES AND
IMPLEMENTATION-DEPENDENT FEATURES ON MIL-STD-1750A ARCHITECTURE

THESIS

Presented to the Faculty of the School of Engineering
of the Air Force Institute of Technology

Air University

In Partial Fulfillment of the
Requirements for the Degree of
Master of Science in Computer Systems

Daniel O. Joyce, B.S.

Captain, USAF

December 1987

* Ada is a registered trademark of the U.S. Government (AJPO).

Approved for public release; distribution unlimited.

Acknowledgments

I would like to thank the people who helped me to turn a one page prospectus into a thesis. Mr. Phil Hanselman, 1st Lt Marc Pitarys, and 1st Lt Robert Marmelstein of the Air Force Wright Aeronautical Laboratory and Capt Dave King and SSgt James Bennett of the Systems Engineering Avionics Facility patiently explained the quirks of the Ada compilers and MIL-STD-1750A computers I used in this thesis, which allowed me to accomplish much more than I could have without their help. I also thank my thesis advisor, Lt Col Richard R. Gross, for forcing me to set my own course in the work, while keeping me from going down too many blind alleys.

Finally, I would like to thank my wife, Gail, for her unflagging support through this entire project. She helped me through the darkest hours, for which I can never thank her enough.

Accession For	
DTIC	<input checked="checked" type="checkbox"/>
DTIC	<input type="checkbox"/>
Unpublished	<input type="checkbox"/>
Classification	
By	
Distribution/	
Availability Codes	
Dist	Special
A-1	

Daniel O. Joyce



Table of Contents

	Page
Acknowledgments	ii
List of Figures	vi
List of Tables	vii
List of Acronyms	viii
Abstract	ix
I. Introduction	1
Problem Statement	2
Background	2
Scope	5
Research Approach	5
Chapter 13 Feature Identification	5
Write and Run Prototype Tests	6
Identify/Obtain Compilers	6
Write and Run Tests	
on Host-Targeted Compilers	6
Run Tests on 1750A-Targeted Compilers	7
Certify Tests and Analyze Results	7
Materials and Equipment	7
Maximum Expected Gain	8
Sequence of Presentation	8
II. Literature Survey	10
Compiler Validation	10
Compiler Performance Evaluation	12
III. System Design of Chapter 13 Feature Tests	18
Identification of Chapter 13	
Features for Testing	18
Address Clauses and Interrupts	19
Record Representation Clauses	21
Length Clauses	21
Interfaces to Other Languages	21
Validation Test Requirements/Design	22
Performance Evaluation Test Requirements/Design	24
Summary	31
IV. Detailed Design of Test Procedures	33
Prototype Test Design	33

Detailed Prototype	
Validation Test Design	33
Detailed Prototype	
Performance Evaluation Test Design	35
Problems in Prototype	
Test Design and Certification	40
Address Clauses and Interrupts Test Design	42
Detailed Validation Test Design	42
Detailed Performance	
Evaluation Test Design	42
Validation Test Design Problems	47
Performance Evaluation	
Test Design Problems	51
Summary	52
V. Analysis and Results	54
Certification of Validation Tests	55
Test Simplicity	55
Minimization of Error-Prone Constructs	55
Certification of Test Correctness	
Without the Feature	56
Combinations Tested Separately	56
Eliminate Implementation-	
Dependencies in Tests	57
Review by ACVC Test Developers	57
Validation Test Results	58
Enumeration Representation Clauses	59
Address Clauses and Interrupts	59
Certification of Performance Evaluation Tests	60
Current Use of Interrupts Feature	61
Alternative Use of Interrupts Feature	61
Performance Evaluation Test Results	64
Enumeration Representation Clauses	64
Address Clauses and Interrupts	65
Summary	66
VI. Conclusions and Recommendations	68
The Problem Revisited	68
Conclusions	69
Recommendations for Future Research	73
Appendix A: Chapter 13 Test Objectives and	
Design Guidelines	75
Appendix B: Validation Test Software	80
Appendix C: Performance Evaluation Test Software	88
Appendix D: Validation Test Results	99
Appendix E: Performance Evaluation Test Results	103

Appendix F: Detailed Description of Time_Package_1750A . .	104
Appendix G: Two-Sample t Test Calculations	115
Bibliography	117
Vita	120

List of Figures

Figure	Page
1. Benchmark Test Loop	26
2. Optimized Benchmark Test Loop	27
3. PIWG Optimization Control Package	37
4. Performance Benchmark Structure	39
5. Example Interrupt Delay Benchmark	46
6. Address Clause Example	48
7. Address Clause Example with Macro	49

List of Tables

Table	Page
I. GET_ALL_TIMES and CLOCK Execution Times	30
II. Interrupts Benchmark Configuration	44
III. Generic Compiler Description	54
IV. Interrupts Benchmark Results	65

List of Acronyms

ACM	Association for Computing Machinery
ACVC	Ada Compiler Validation Capability
ACEC	Ada Compiler Evaluation Capability
AFWAL	Air Force Wright Aeronautical Laboratory
AIG	ACVC Implementers' Guide
AJPO	Ada Joint Program Office
ASD	Aeronautical Systems Division
AVF	Ada Validation Facility
CPU	Central Processing Unit
DoD	Department of Defense
HOL	High Order Language
KB	Kilo Byte
I/O	Input/Output
LRM	Language Reference Manual
PIWG	Performance Issues Working Group
SIGAda	Special Interest Group on Ada

Abstract

Developers of applications for embedded systems need full implementations for all of the representation clauses and implementation-dependent features in Chapter 13 of the Language Reference Manual (LRM) if they are to be successful in developing these applications entirely in Ada. Because implementations of Ada's representation clauses and implementation-dependent features vary from compiler to compiler, these features must be validated and evaluated before they are used in applications that have such high reliability requirements. This thesis describes an approach used to develop validation tests and performance evaluation tests, or benchmarks, for Ada's address clauses and interrupts features and reports the results of the validation tests and benchmarks.

The validation tests were compiled with three validated Ada compilers, two of which were targeted to the MIL-STD-1750A processor. The benchmarks developed in this research measure interrupt delay time for interrupts associated with a task entry by an address clause. These benchmarks were compiled with a validated Ada compiler targeted to the MIL-STD-1750A and run on a Sperry 1631 MIL-STD-1750A processor.

VALIDATING AND EVALUATING ADA'S REPRESENTATION CLAUSES AND IMPLEMENTATION-DEPENDENT FEATURES ON MIL-STD-1750A ARCHITECTURE

I. Introduction

In the mid-1970s, because of rising software costs and increasingly unreliable software, the Department of Defense (DoD) started the development of a new high-order computer programming language, now known as Ada. In an effort to direct the development and use of the language, the DoD established the Ada Joint Program Office (AJPO) to manage all Ada-related activities (Booch, 1987:22). One of the most important activities of the AJPO is validating Ada compilers (programs that translate Ada source code into machine instructions), because, by DoD directive, only *validated* Ada compilers can be used to develop Ada software for the DoD (AJPO, 1987). To be validated, a compiler must pass a series of tests, known as the Ada Compiler Validation Capability (ACVC), that demonstrates the compiler's adherence to ANSI/MIL-STD-1815A, Ada Programming Language Reference Manual (LRM) (DoD, 1983).

Even if a compiler is validated, it still may not meet the execution requirements of embedded systems. The validation process does not provide an evaluation of a compiler's efficiency or performance; it will only determine whether or not a compiler complies with the Ada LRM (AJPO, 1987:1).

Problem Statement

Before using an Ada *representation clause* or implementation-dependent feature described in Chapter 13¹ of the LRM, researchers and developers of embedded applications in Ada must determine two things:

- 1) Does the Chapter 13 feature's implementation conform to the description in the Ada Programming Language Reference Manual (LRM)?
- 2) If so, how efficient is the implementation (compared to other compilers' implementations of the same feature)?

Ada validation and performance evaluation tests did not exist to answer these questions when this thesis project began.

Background

One of the driving forces behind the development of Ada was a Department of Defense (DoD) need to develop a high order language that met the requirements of embedded systems (Booch, 1987:14). Booch defines an embedded system as a 'computer system . . . that forms a part of a larger system whose purpose is not primarily computational, such as a weapons system or a process controller' (Booch, 1987:15-16). Because of physical limitations on embedded systems' memory (often 64 to 128 kilobytes) and embedded systems' real-time processing requirements, applications for embedded systems often were (and still are) written in JOVIAL (DoD, 1984), assembly

¹ For the sake of brevity I will refer to the representation clauses and implementation-dependent features of Ada as 'Chapter 13 features' in the course of this thesis.

language, or some combination of the two (King, 1987), because of the compactness and execution time efficiency of applications developed in these languages. However, although JOVIAL and assembly language routines can address memory directly and efficiently perform other low-level operations, applications written in these languages are difficult to maintain. Since the DoD has directed that Ada be used to develop all new embedded applications (DoD, 1987), the Chapter 13 features will have to perform these low-level operations as efficiently as JOVIAL and assembly language if Ada is to eventually replace those languages.

Ada's representation clauses (1) describe how data types (such as integer, real, array, and record types) are to be mapped onto the underlying machine, thus allowing for a more efficient representation than would be possible with the default definitions for these types; and (2) allow the direct specification of memory addresses for objects and programs (DoD, 1983:Ch 13, 1). For example, assume an embedded system whose normal representation for integer data types is a 16-bit word. If the developer knows that all of the integer values in an application can be stored in eight bits, he can define the representation for integers to be eight bits, effectively doubling the number of integers he can store.

The implementation-dependent features of Ada provide (1) for specifying underlying system characteristics; (2) for freeing unused memory that has been dynamically allocated; and (3) for performing data conversions that Ada normally does not

allow because of its strong typing (Booch, 1987:332). Without these features, Ada can not perform the same functions as assembly language routines; therefore, the features in Chapter 13 are vital to any embedded applications written entirely in Ada.

Only one organization, SofTech Inc., is now developing tests (Wilson, 1987a) for the ACVC for the existence and correct implementation of Chapter 13. While additional tests will be added to the ACVC in subsequent releases, version 1.9 of the ACVC, released 1 June 1987, included only four Chapter 13 tests, and these test but one feature (Wilson, 1987a).

There are at least two major efforts underway for evaluating the performance of Ada compilers. The Boeing Military Airplane Company is currently under contract to the Ada Evaluation and Validation Team of the AJPO to develop the Ada Compiler Evaluation Capability (ACEC), a test suite that will include performance tests for (among others) Chapter 13 features of the language (BMAC, 1987). The second effort is that of the Performance Issues Working Group (PIWG) of the Association for Computing Machinery (ACM) Special Interest Group on Ada (SIGAda). Among other activities, the PIWG collects compiler benchmarks from its members and places them in the public domain; performance evaluation benchmarks for Chapter 13 features are among these (Squire, 1987).

Scope

The goals of this thesis project, then, were (1) to expand the existing compiler validation and evaluation methods or develop new methods, as necessary, for validating and evaluating Chapter 13 features of Ada and (2) to certify these methods by writing needed validation and evaluation tests and applying them to compilers targeted to embedded computers. All compilers run on one computer (the 'host computer') and produce machine instructions for another computer (the 'target computer') that may or may not be the same as the host. In this thesis, I studied Ada compilers targeted to MIL-STD-1750A computers, the standard airborne (embedded) computers for the U.S. Air Force (DoD, 1982:1).

I did not develop tests for all of the features in Chapter 13; instead, I concentrated on the features most important to the development of embedded systems applications. My ordering of importance was based on the assumption that embedded systems developers would be using a mix of Ada, JOVIAL, and assembly language in the near future because of the current lack of Ada compilers with full Chapter 13 implementations targeted to the MIL-STD-1750A computer.

Research Approach

Chapter 13 Feature Identification. I identified the most important Chapter 13 features for embedded systems by a review of current literature and by discussions with embedded systems developers in the Aeronautical Systems Division Systems

Engineering Avionics Facility (ASD/ENASF), embedded systems developers in the Air Force Wright Aeronautical Laboratories System Evaluation Branch (AFWAL/AAAF) and Information Processing Technology Branch (AFWAL/AAAT), and other embedded systems developers in the Air Force and industry.

Write and Run Prototype Tests. I wrote tests for the validity and efficiency of a Chapter 13 feature that was implemented by at least two Ada compilers targeted to the host computer. The process of writing and running these tests helped to refine the requirements for the remaining tests, to identify potential problem areas, and to learn the size and difficulty of the thesis problem.

Identify/Obtain Compilers. Based on the Chapter 13 features identified for study, I chose the candidate compilers that were used in my thesis project. I included compilers that were hosted on and targeted to the same computer, as well as compilers that were targeted to a MIL-STD-1750A computer to provide a broad range of compiler/host combinations. Compilers targeted to the host machine typically included better tools (such as debuggers), which were helpful in developing the test software.

Write and Run Tests on Host-Targeted Compilers. Using the methods developed in writing the prototype tests, I wrote the tests for the Chapter 13 features identified earlier and then ran the tests on the compilers that were targeted to the host computer to take advantage of diagnostic tools discussed

above. Based on this preliminary testing and analysis, I had to re-write and re-run some of these tests.

Run Tests on 1750A-Targeted Compilers. After establishing the validity of the tests on the host-targeted compilers, I applied them to the Ada compilers that are targeted to the MIL-STD-1750A computers.

Certify Tests and Analyze Results. Finally, I certified that the tests developed in this thesis answered the questions stated in the thesis problem and analyzed the results of the tests. The results of the validation tests were discrete; a compiler either passed or failed the applicable tests. The results of the performance evaluation test were continuous execution times, so I was able to support the hypothesis that the mean test execution times for various benchmarks differed, and to explain the causes for those differences. Using these results, I wrote conclusions and recommendations.

Materials and Equipment

The compilers needed for this thesis, including compilers that are targeted to the MIL-STD-1750A computer, were generally available at AFIT and through organizations at Wright-Patterson AFB. AFWAL/AAAT and the thesis sponsor, ASD/ENASF, provided access to MIL-STD-1750A computers, Ada compilers targeted to those computers, and the assistance of personnel knowledgeable in the operation of this hardware and software. Finally, I was given access to the ARPANET to obtain public domain test software and information on the state of current Ada research.

Maximum Expected Gain

I expected this thesis to break new ground in the area of testing Ada's Chapter 13 features. Many are hesitant to use Ada because it is an immature language, and few want to take a risk on unproven features in the language (Myers, 1987:71). This thesis should remove some of the uncertainty surrounding Chapter 13 features by providing sound tests for the validity and performance of those features. This work is immediately useful to developers of embedded applications in Ada, as well as to those who are developing Ada compilers that will implement these features, because it provides a method for testing the validity and efficiency of these features before they must be used. Finally, the AVF expressed an interest in the results of this thesis in developing new tests for the ACVC test suite (Chitwood, 1987), and some of the validation tests written in this thesis were incorporated in version 1.10 of the Ada Compiler Validation Capability test suite (Brashear, 1987b).

Sequence of Presentation

The theory and method of compiler validation and performance evaluation are summarized in Chapter 2. This chapter also identifies some of the deficiencies of previous methods in these two fields.

The design of the validation and evaluation tests developed in this thesis is presented in Chapter 3. This chapter describes the test requirements, justifies the

requirements, and gives a general approach to test design that fulfills these requirements.

Chapter 4 describes the detailed design of the tests.

Chapter 5 details how the tests were certified and reports the results of the Chapter 13 feature tests.

The conclusions of this project and recommendations for future research are presented in Chapter 6.

II. Literature Survey

The development of tests for correct and efficient implementations of features of the Ada programming language is guided by two of the most important disciplines of compiler analysis: validation and performance evaluation. This chapter will review current theory and method in these two areas and relate them to the problem of testing representation clauses and implementation-dependent features of Ada.

Compiler Validation

One of the major design requirements for Ada in the early stages of its development was that the software developed with Ada and the programmers trained with Ada be portable (Ploedereder, 1986: Ch 7, 1). All too often, pre-Ada software had to be extensively rewritten when moved to another computer system, and programmers had to be trained in new programming languages whenever they moved. Although standards exist for languages such as Pascal, JOVIAL, and COBOL, past experience had shown that software was often developed using a non-standard or extended-standard compiler, making it very costly to move an application to a new computer because of the differences between the old and new compilers. This is one of the primary reasons that no subset, extended, or non-standard Ada compilers are allowed (Ploedereder, 1986: Ch 7, 1).

The Ada Compiler Validation Capability (ACVC) Implementers' Guide "describes implementation implications of the LRM and conditions to be checked by validation tests"

(Goodenough, 1986:Ch 1, 1). This guide provides semantic ramifications, legality rules, and AJPO-approved interpretations of Ada constructs as well as test objectives and design guidelines for all validation tests. Although test objectives and design guidelines have been written, only a subset of these have been turned into working validation test programs (Wilson, 1987a).

The 3100 tests in version 1.9 of the ACVC fall into the following classes: (1) Class A tests ensure the minimum allowable set of the language is implemented; (2) Class B tests determine whether a compiler can detect illegal uses of the language; (3) Class C tests are run-time tests that should compile and execute successfully; (4) Class D tests determine the capacity of the implementation of the language constructs, such as determining how many levels of loop nesting an implementation will support; (5) Class E tests determine whether implementation-dependent attributes of language features have been provided for; and (6) Class L tests determine the compiler's ability to detect link-time errors (Wilson, 1987b). A compiler, however, does not necessarily have to implement all of the features in Chapter 13 to pass these tests. For example, the Ada LRM states: "An implementation may limit its acceptance of representation clauses to those that can be handled simply by the underlying hardware" (DoD, 1983: Ch 13, 2). Because certain Chapter 13 features are considered optional and because initial ACVC emphasis was placed on testing the mandatory features of the language, early

versions of the ACVC did not include tests for features in Chapter 13 of the LRM, even though some such features, such as package SYSTEM (which defines characteristics of the target computer) *must* be provided by an implementation (DoD, 1983:Ch 13, 9).

Also, because Ada compilers are currently validated with a test suite that tests only a subset of the language (Wilson, 1987a), a validated compiler could have an illegal implementation of some untested feature. For example, a compiler that has an incorrect implementation for address clauses (one of the Chapter 13 features) could be validated with the current (version 1.9) or previous versions of the ACVC test suite because these versions did not contain tests for address clauses.

Finally, because the ACVC test suite makes no inferences about the relative efficiency of one compiler versus another (Wilson, 1987a), compiler validation cannot be used to determine the suitability of a compiler for a particular application (AJPO, 1987:1). The Ada compiler validation process has discrete results -- either a compiler passes all applicable tests and is validated as conforming to the standard, or it fails at least one applicable test and is not validated.

Compiler Performance Evaluation

Compiler performance evaluation, or *benchmarking*, is an outgrowth of the computer benchmarking field. Computer

benchmarking techniques developed in the early 1970s, as potential purchasers of large computer systems searched for a tool that could provide an objective measure of a computer's processing power (Benwell, 1975: vii-viii). Performance evaluation programs, or *benchmarks*, were developed to measure a computer's processing power by measuring the computer's execution time for a representative set of computer instructions. The basic approach in benchmarking computers is to hold all other factors (such as computer programming language use, number of iterations, etc.) constant in a benchmark execution, varying only the computer being benchmarked; thus any difference in test results may be attributed to differences in the computers' execution speeds. This same method may be applied to benchmarking compilers.

There are three approaches that have been used to evaluate the efficiency of Ada and other high order languages (HOLs) using the metric of execution speed:

writing (1) a set of small well-established numerical benchmarks, (2) a sample of representative programs from the application domain, and (3) a synthetic benchmark in Ada and other high order languages, viz., FORTRAN and Pascal, and comparing the resulting compilation and execution times (Bassman and others, 1985:151).

Numerical benchmarks were originally developed to benchmark scientific computers whose primary functions were floating point mathematical operations. Because such benchmarks do not typically use the more modern constructs, such as tasking and access types, of a language such as Ada, they cannot accurately evaluate these features (Weicker, 1984:1013). *Application*

domain benchmarks are application programs that have been modified to include code for reporting performance of the program (Craine, 1986:13). Because compilers targeted to MIL-STD-1750A computers have so far implemented few of the Chapter 13 features, applications using these features are generally unavailable. A *synthetic benchmark* includes a balance of instructions that is typical of the general use of a computer language, based on a statistical analysis of a large number of applications written in that language (Bassman and others, 1985:151).

While these three approaches provide comparative information for general-purpose (non-embedded) computers, they do not give quantitative results that can be used by the compiler implementers and users concerned with the effects of specific language features, because the results are distorted by the effects of other language constructs used in the benchmarks (Bassman and others, 1985:151).

To benchmark the performance of an Ada compiler and that of another HOL such as Pascal (or another vendor's version of Ada), another approach that may be adapted to embedded computer systems includes:

- (1) Programming test cases for fundamental language features in Ada, the HOL, and assembly language; (2) Writing test cases for code improvement in Ada and the HOL; and (3) Programming representative applications in Ada, the HOL, and assembly language (Bassman and others, 1985:153).

This approach is suitable for benchmarking compilers targeted to embedded processors because it provides detailed knowledge

of the performance of individual features' (Clapp and others, 1986:767; Bassman and others, 1985:155). In this approach, the benchmarks are designed to measure the execution time of specific language features by isolating the feature in a test loop of the benchmark and determining the difference in execution time between the test loop and a control loop in which the feature is not used (Clapp and others, 1986:767). This approach will be referred to as the 'dual-loop approach' in the remainder of this thesis.

Because of the limited memory available to and real-time processing requirements for embedded applications, two of the primary requirements for benchmarks for Chapter 13 features in embedded applications are that the benchmark measure the feature's *memory use* and *execution speed* (Phillips and Stevenson, 1984:4.100). With respect to memory requirements, the MIL-STD-1750A computer with the extended memory option now allows an application to address up to one megaword of memory, although only 64K words each of data and instructions may be accessed without changing page registers (Bunce, 1987). As long as program units (such as tasks) can be designed so that they stay within a 64K boundary accessible within the logical address space, the application does not suffer from excessive page swapping (Bunce, 1987). New embedded applications being developed for MIL-STD-1750A will most likely use the extended memory option, lessening the size restrictions on those applications (Lyons, 1987). One researcher further found that the arguments against using Ada for applications targeted to

the MIL-STD-1750A computer because of insufficient memory were based on worst-case scenarios instead of the class of applications more likely to be found on embedded systems (Roark, 1987b). In fact, embedded real-time applications researchers and developers interviewed in this research were able to work around memory limitations or found that their applications were not limited by the larger memory available with the extended memory MIL-STD-1750A (Clements, 1987; Roark 1987b). Nevertheless, embedded systems benchmarks should still address memory usage as a concern.

With respect to time requirements, although research is currently being conducted to develop faster MIL-STD-1750A computers using Very High Speed Integrated Circuit (VHSIC) technology, this technology may not be available in time to solve the more pressing timing restrictions currently placed on embedded applications (Pitarys, 1987). Furthermore, history indicates that time efficiency will remain a concern even if much faster processors are found.

Currently there are two major sets of performance evaluation software available in the public domain: the SIGAda PIWG benchmarks and the Prototype Ada Compiler Evaluation Capability (ACEC). The SIGAda benchmarks are a collection of benchmarks that use the dual-loop approach to measure the execution time associated with such constructs as task creation and elaboration, exception handling, package TEXT_IO, loops, procedure calls, various task design styles, and packed boolean arrays (PIWG, 1987). Except for the benchmarks that measure

the effects of packing boolean arrays, none of these benchmarks measures any Chapter 13 features.

The prototype ACEC benchmarks are a collection of benchmarks from the public domain, organized for the Evaluation and Validation team of the AJPO by the Institute for Defense Analyses. These include numerical and synthetic benchmarks that measure the execution time associated with tasking, procedure calls, loops, case statements, recursive calls, and global variable access, among others. The prototype ACEC does include benchmarks for the use of Chapter 13 features, including measurements for unchecked storage deallocation and mathematical operations on objects declared with a length clause (Witt, 1985:90-93).

Other sets of benchmarks available are those collected on the Ada Software Repository (Conn, 1987:130) which include numerical and synthetic benchmarks from the 1985 Los Angeles ACM AdaTEC conference; other numerical benchmarks such as Whetstone, Dhrystone, and the Sieve of Erasthones; synthetic benchmarks to measure tasking overhead; and finally, a copy of the ACM SIGAda PIWG benchmarks. These collections do not contain any new benchmarks for Chapter 13 features beyond those described previously in this chapter.

III. System Design of Chapter 13 Feature Tests

This chapter identifies the Chapter 13 features chosen for evaluation and describes why they are more important to embedded applications than other features. The requirements and justification are also presented for the design of a system of tests and procedures to validate and evaluate Chapter 13 features.

Identification of Chapter 13 Features for Testing

Recall from Chapter 1 that the purpose of this research was to expand the current validation and performance evaluation techniques to develop tests to validate and evaluate the features of Chapter 13 most important to embedded applications. This section describes the Chapter 13 features that were chosen and explains why they were chosen.

Because there are currently very few Ada compilers with full Chapter 13 implementations available, most of the research to identify the Chapter 13 features important to embedded applications developers was conducted through interviews with those developers. Some embedded applications developers interviewed stated flatly that they needed most or all of the Chapter 13 features for their applications (Roark, 1987; Shaw, 1987; Lyons, 1987). Others were able to prioritize the need for implementation of certain features based on their knowledge that they currently used (or would soon use) the feature in an application, or conversely, that they would not use a feature (because they had no need for it, the feature was too risky to

use, or the capability could be provided in other ways) (King, 1987; Bennett, 1987; Seward, 1987). The lack of production-quality compilers targeted to MIL-STD-1750A computers still forces developers of these applications to use a mixture of Ada, assembly language, and JOVIAL (Bennett, 1987).

In these discussions with embedded applications developers and compiler designers for those applications, then, the features that were identified as most important were Address Clauses and Interrupts, Record Representation Clauses, Length Clauses, and Interface to Other Languages.

Address Clause and Interrupts. Ada's address clauses allow a developer to specify the memory location (or address) at which a variable is to be stored or at which a subprogram or package is to begin. They may also be used to link an interrupt to an Ada task that will handle the interrupt (DoD, 1983:Ch 13, 5). These features are vital to embedded applications, especially those that seek to minimize the number of assembly language routines included to perform low-level operations. It is the nature of embedded applications that certain operations, such as reading data from pre-determined Input/Output (I/O) ports, must be tailored to the configuration of the hardware. Because decisions regarding the physical location of registers and I/O ports on the underlying hardware are made long before software is developed, software must adapt to the hardware. For example, the device that determines the airspeed of an aircraft may be physically connected to an I/O port at memory address 2000 of an embedded computer, so data

from the device must be read from that location. By specifying that the variable that will contain the information be loaded at that address, the data from the device can be read from the variable. If this feature were not provided, an application would have to make a call to an assembly language or JOVIAL routine to retrieve the data from address 2000.

Another feature that is closely related to address clauses is Ada's interrupt handling feature. An *interrupt* is an event during the execution of a program that indicates an error in the hardware or software has occurred or that signals that some device attached to the computer needs service. The interrupt notifies the Central Processing Unit (CPU) to suspend its current operation and take action on, or *handle*, the event that caused the interrupt. For example, an I/O interrupt may signal the CPU that data is ready to be read from an I/O port. The CPU, in turn, handles the interrupt by moving the data from the I/O port to main memory. Many embedded applications, such as avionics, are typically *interrupt driven* (DoD, 1982), i.e., interrupts (rather than polling schemes) are used to indicate when I/O or hardware devices need service (Bennett, 1987). Since Ada was designed for embedded systems (Booch, 1987), providing a capability to handle interrupts is crucial. Ada's interrupts feature allows applications to handle asynchronous interrupts by associating a task entry with a memory location that contains the interrupt vector or the number of the interrupt itself. When the interrupt occurs, the associated

task entry is called and the interrupt is handled by the body of the task.

Record Representation Clauses. Record representation clauses allow the embedded applications developer to declare and use variables that represent low-level values normally not accessible with other high-order languages. A common example of this would be to use the record representation clause to declare a record representation for a computer's program status word (PSW) (DoD, 1983:Ch 13, 6; DoD, 1982, 11), in which information about the state of the CPU and other devices is stored in specific bit positions. The capability to access the PSW can be vital to an application because its values often determine the sequence of future operations in the application. Without record representation clauses, an Ada application would have to incorporate a number of assembly language routines to retrieve and decode information stored in registers such as the PSW (Seward, 1987).

Length Clauses. Length clauses, like the features described earlier, allow the developer to tailor his application to the underlying hardware. They take advantage of physical characteristics (such as word length) to define for data types more efficient representations than the defaults (DoD, 1983:Ch 13, 3). Because of the execution time and storage space requirements placed on many embedded applications, length clauses are thus necessary (Seward, 1987).

Interfaces to Other Languages. Because embedded applications in Ada still currently need to call assembly and

JOVIAL routines to perform certain low-level operations (Bennett, 1987), without this feature, Ada could not now be used for embedded applications.

Other features in Chapter 13 were not studied because they could be worked around more easily or were too risky to use in embedded applications (Bennett, 1987; Roark, 1987; Seward, 1987). *Change of representation*, a feature that allows the developer to specify an alternate representation for a data type, was not studied because its function could be provided (albeit less efficiently) by explicit type conversions and declarations of additional data types. *Machine code insertions* allow the developer to place assembly language instructions directly in a procedure, but the same result may be obtained by calling an assembly language routine with the interface to other languages feature. *Unchecked storage deallocation*, used to free for further use memory that had been previously dynamically allocated, was not studied because the developers either do not have the high confidence in the correct operation of the feature, or cannot make the required determination of the maximum dynamic memory requirements, or both. (Bennett, 1987; Seward, 1987). Finally, *unchecked type conversion* was not studied because it would not be used as often as the other features identified for study (Bennett, 1987).

Validation Test Requirements/Design

The primary design requirement for the validation tests was that they determine whether a compiler complies with the

Ada LRM (AJPO, 1987). This requirement is expanded in the ACVC Implementers' Guide (AIG) (described in Chapter 2) in specific design guidelines for tests for the entire Ada language (Goodenough, 1986). In particular, Ada validation tests must clearly state their objectives, identify the language feature being tested, and generate specific information on whether the compiler passed or failed the test. For such generation, my validation tests used the standard ACVC test reporting package REPORT that wrote the results of the tests using Ada's package TEXT_IO². The reasons for this were twofold: by designing for re-usability of existing software, I was able to spend more time developing and testing my design; and this approach made the tests I developed reusable by organizations such as the Ada Validation Facility (AVF). The validation tests were likewise categorized according to the six ACVC test classes described in Chapter 2.

While validation tests provide only a *yes* or *no* answer for implementations of a Chapter 13 feature, these tests are a prerequisite for evaluating the performance of the feature. An efficiency test comparing two compilers may have no meaning if one of the compilers being tested has an invalid implementation of the feature in question.

A compiler either implemented a feature correctly and passed all validation tests, or it failed one or more of the

² TEXT_IO can be used for this purpose, because all Ada compilers must implement TEXT_IO to pass a compiler validation (Wilson, 1987a).

tests and was ruled non-valid. Therefore, statistical methods could not be applied to the validation tests to determine their validity. The validity of the tests was determined instead through reviews of the tests by ACVC test developers in the Ada Validation Facility and at SofTech, Inc. These two organizations are responsible for the development and maintenance of all tests in the ACVC test suite (Wilson, 1987b). These reviews, identical to those conducted to certify ACVC tests, ensured that test design objectives were fulfilled and that the tests contained no obvious errors.

Performance Evaluation Test Requirements/Design

The two metrics described in chapter 2 that were used to determine the efficiency of Chapter 13 features in embedded applications were *memory use* and *execution speed*. Additional requirements for benchmarks for these language features are:

1. The features to be measured must be isolated and compiler optimizations that would invalidate the measurement must be avoided.
2. Sufficient accuracy in the measurement must be obtained.
3. Operating system distortions must be avoided.
4. The results obtained must be repeatable (Clapp and others, 1986:760-761).

As the research progressed, it became apparent that the time metric was more important in determining the efficiency of Chapter 13 features, because the MIL-STD-1750A extended memory option allows an application to address up to one megaword of memory, reducing the space constraints on many embedded applications. Therefore, this thesis concentrated on benchmarks that measured the execution time associated with the

use of a Chapter 13 feature as a metric for the efficiency of that feature's implementation.

The design of a test of the performance of a Chapter 13 feature was based on the 'dual-loop' method of Clapp and others described in Chapter 2. This approach concentrates on measuring the effect of using one feature and meticulously negates all other factors that may affect the timing results. This is the best method for determining whether a compiler's implementation of a feature is suitable for an embedded real-time application, because it alone (as stated in chapter 2) provides specific comparative information about the effects of the use of that one feature. Furthermore, it is the same method used in many of the benchmarks in the ACM SIGAda PIWG test suite (PIWG, 1986). By reusing PIWG software for timing measurements, data collection, and test reporting, I had more time to develop and validate the tests for performance evaluation of Chapter 13 features. Numerical, application domain, and synthetic benchmarks were not used because they could not provide comparative measures of compiler efficiency for specific language features.

As described in Chapter 2, benchmarks for Chapter 13 features had to meet additional requirements. The *isolation of the test feature* was accomplished by writing two parallel sets of executable operations, one set that used the feature and one that did not. Each of these sets was placed in an identical loop so that the only difference between the two loops was the use of the feature being studied.

Avoiding compiler optimization of the tests is critical because optimization can radically alter the nature of the test. Consider the test loop in figure 1.

```
Test_Start_Time := Current_CPU_Time;

For I = 1 to 100 loop
    Test_Variable := 10;
end loop;

Test_Stop_Time := Current_CPU_Time;
Test_Total_Time := Test_Stop_Time
                  - Test_Start_Time;
```

Figure 1. Benchmark Test Loop

The objective of this test is to measure the time it takes to perform an assignment statement 100 times in a loop. The Test_Start_Time and Test_Stop_Time variables take time readings before and after the loop from the function Current_CPU_Time, and the total time for the test is determined by their difference. Because the value of Test_Variable will be 10 no matter how many times the loop is executed, an optimizing compiler may generate executable statements that will perform the operations shown in figure 2. In this optimized version, the assignment has been moved out of the loop by the compiler to make the program more efficient. Some compilers may even remove the loop entirely, replacing it with the Test_Variable assignment statement. The segment of code in figure 2 has the same logical result (i.e. Test_Variable is set to 2) as the segment of code in figure 1, but the results of the execution time measurement may be very different.

```

Test_Start_Time := Current_CPU_Time;

For I = 1 to 100 loop
    null;  -- Do nothing in the loop
end loop;

Test_Variable := 10;

Test_Stop_Time := Current_CPU_Time;
Test_Total_Time := Test_Stop_Time
                  - Test_Start_Time;

```

Figure 2. Optimized Benchmark Test Loop

Some compilers allow the user to control optimization by setting a parameter for the compilation, instructing the compiler to perform no optimization on the instructions. Another method is to keep the values of objects in the loop hidden from the compiler by placing them in a procedure that is compiled after the test program. This method was used in this thesis because it ensured no optimization, whether an optimization control switch was set or not.

The *accuracy of the time measurement* is a function of the resolution of the clock available on the computer on which the benchmark is run. A benchmark that is generic enough to run on a number of compilers must use the Ada CLOCK function in package CALENDAR, because it is the only function that all Ada compilers must implement that returns the current CPU time of the underlying hardware (DoD, 1983:Ch 9, 11). Such use of the Ada CLOCK function presents a number of timing anomalies that must be accounted for when designing a benchmark: insufficient

clock precision, variations in the clock, and clock overhead (Altman, 1987:11-16).

The CLOCK function in Ada does not provide a continuous representation of time: rather, it expresses the time in increments t , with t defined by the value SYSTEM.TICK (DoD, 1983:Ch 13, 11). The execution time of a benchmark therefore will always return a time of some integral multiple of this increment, $n * t$. Because the actual time a benchmark takes to execute may not be an exact multiple of t , the actual time will lie between $n * t$ and $(n + 1) * t$. The smaller n is, the larger the variability of the benchmark's results. By increasing the number of the loop executions, this variability can be reduced and a more accurate time for the effect of the feature can be produced by averaging the time for one execution (loop) of the benchmark (Clapp and others, 1986:762); this was the method used in this thesis.

In the course of this research, I found that the CLOCK function did not return time values with the precision necessary to measure the events associated with the features I was studying. I then wrote package TIME_PACKAGE_1750A, which includes procedure GET_ALL_TIMES, designed to retrieve the time from the hardware clock registers of the MIL-STD-1750A processor. This package and its development are described in detail in Appendix F.

Although GET_ALL_TIMES returns CPU times with greater precision than those from the CLOCK function in package CALENDAR, it too is affected by the discrete-function timing

anomalies described above. Furthermore, a clock may *drift*, or gradually run slow or fast compared to an ideal clock; or a clock may *jitter*, speeding up or slowing down briefly, but remaining accurate over longer time spans (Altman, 1987:12-15). Without performing tests on the computer hardware with another clock, one cannot detect drift and jitter but should be aware that they may occur. Jitter in a clock will tend to be offset as the number of loop iterations increases and longer time spans are used to record the results of the benchmark. One approach to negate the effects of drift is to repeat the benchmarks a number of times, so that the drift of a clock would apply equally to the test and control loops of a benchmark, still providing a comparative measure of compiler performance. The alternative to this is to measure the drift of the clock with tests on the computer hardware (e.g., connecting an external clock to the computer and measuring the differences in the two clocks), or to use an external clock to record benchmark timing data (Altman, 1987:16). Detecting the drift of a clock, if it existed, was beyond the scope of this thesis. The benchmark executions were repeated, however, so that if drift were present, its effect would be negated as described above.

Overhead in the GET_ALL_TIMES calls was negated by making identical calls for a test and control loop in the benchmark, so that when the elapsed time was calculated by subtracting the clock start time from the clock stop time, the overhead

introduced by the GET_ALL_TIMES call was eliminated (Altman, 1987:16).

As a mechanism for verifying that the GET_ALL_TIMES procedure was providing an accurate measure of the time required for the test execution, GET_ALL_TIMES and the Ada CLOCK function were used to time identical loops that performed an assignment, addition, and conditional (*if-then-else*) statement. Techniques for control of optimization, described earlier, were used in the design of the loops, thus the execution times should have been identical. Two separate programs were written, identical except for the routine used to retrieve the current CPU time. The programs executed the loop 30,000 times and were both executed 50 times on a Sperry 1631³ MIL-STD-1750A. The elapsed loop time was found by subtracting the overhead of the CPU time retrieval calls from the elapsed loop time. The results are reported in table 1 below.

Table 1
GET_ALL_TIMES and CLOCK Execution Times

Time Retrieval Program	Mean (seconds)	Standard Deviation (seconds)
GET_ALL_TIMES	0.55158	0.00028
Ada CLOCK	0.52510	0.00317

³ Use of this processor does not necessarily indicate the compilers studied in this thesis. The Sperry 1631 processor executed test programs compiled with at least three validated Ada compilers targeted to the MIL-STD-1750A during my research.

The variability in the results may be attributed to the reduced precision available with the CLOCK function. Package CALENDAR reports all times using a fixed point data type name DURATION, which stores time data accurate to 200 microseconds. The GET_ALL_TIMES procedure can retrieve and store time data accurate to 10 microseconds. Given that GET_ALL_TIMES reports current CPU time more accurately and precisely than the function available in package CALENDAR, GET_ALL_TIMES was used in the remainder of the research for CPU time retrieval.

The third additional requirement of the dual-loop method, eliminating *operating system distortion*, is not a major concern with the MIL-STD-1750A computer. This requirement was included principally for time-sharing or multiprocessing systems in which other processors or system functions may interfere with the operation of the benchmark (Clapp and others, 1986:763). With the MIL-STD-1750A, such distortion was eliminated by running the benchmark and nothing else on the computer.

Finally, *repeatability of test results* was provided in the design by making no assumptions about pre-conditions for the benchmarks. The benchmarks were designed to run alone on a MIL-STD-1750A, therefore all benchmark executions will report identical results.

Summary

This chapter identified the most important features in Chapter 13 for study and described why they are important. The requirements for validation and performance evaluation tests

for these features were presented, followed by a description of how the test design was based on the requirements. In the next chapter, the system design presented here is expanded to give more details of the design process.

IV. Detailed Design of Chapter 13 Feature Tests

This chapter provides more details of the design of the validation and performance evaluation tests for: (1) the prototype Chapter 13 feature, enumeration representation clauses, and (2) the other Chapter 13 features identified as important to embedded applications earlier in this thesis, address clauses and interrupts.

Prototype Test Design

The enumeration representation clause was chosen as the feature for which to write prototype validation and performance evaluation tests, or benchmarks, because of its current and potential use in embedded applications (Hanselman, 1987). The purpose for writing prototype tests, stated in the introduction, was to develop a greater understanding of the requirements for and the problems associated with design and implementation of the tests. The enumeration representation clause was ideal for a prototype because the set of other language features it affected was rather small, which made tests for the validity and performance of the feature much easier to develop. Tackling a larger or more extensive feature first would have consumed more research time for the amount learned delaying further implementation of tests on other Chapter 13 features.

Detailed Prototype Validation Test Design. The detailed design guidelines for the validation tests for enumeration representation clauses came from the ACVC Implementers' Guide

(AIG), as discussed in Chapter 3. The intent in the design of the prototype validation tests was to develop and execute a representative sample of tests that would be required for a complete validation of the feature. The purpose, again, was to learn as much about the process of developing validation tests and the problems associated with this development as quickly as possible. For that reason, four representative tests were written for the 13 test objectives outlined in the AIG for enumeration representation clauses (Goodenough, 1986:Ch 13, 39-40). These tests were selected because they tested the feature as it would most commonly be used. Tests named BD3001A, BD3002A, BD3004A, BD3012A were developed to test objectives 1, 2, 4 and 12 for enumeration representation clauses. The names of these tests conform with the naming conventions for the ACVC test suite provided by the Ada Validation Facility (Wilson, 1987a), making this work more understandable to and usable by those familiar with the ACVC. A description of the ACVC test objectives for validation tests in this thesis is found at Appendix A; a description of the test naming conventions, sample source code for the tests, and instructions on how to obtain machine-readable copies are at Appendix B.

All of the tests developed were Class B tests: that is, a compiler with a valid implementation of this feature should generate compile-time errors for the test, because of the test's illegal use of the enumeration representation clause. Class B tests were developed because a majority of the test objectives for enumeration representation clauses in the AIG

are written to detect illegal uses of the clause. Class A and class C tests could have been written for other test objectives to ensure that the compiler allowed correct uses of the feature.⁴

All of my validation tests, both prototype and operational, were originally designed to include a set of Ada language statements in a separate compilation unit that would implement all of the requirements specified in each test objective of the AIG. The purpose was to keep the number of test compilation units down, providing a more compact and usable test suite. It later became apparent, however, that separate tests had to be written for many of the sub-objectives of a single test objective. The prototype validation tests tested one objective per compilation unit. The operational validation tests included multiple test programs per objective because of restrictions that compilers could place on acceptance of the Chapter 13 feature.

Detailed Prototype Performance Evaluation Test Design.

Following the system design objectives outlined in Chapter 3, I designed a test that (1) isolated the enumeration representation clause and would not be affected by

⁴ In retrospect, it was a mistake not to write more tests for these other test classes. I assumed they would be no different than other tests, so I didn't write any. In that assumption I was wrong; I realized when I wrote executable tests for address clauses that implementation may restrict how the feature may be used.

optimization; (2) would be accurate; (3) avoided operating system distortions; and (4) would be repeatable.

The enumeration representation clause was isolated by declaring two identical enumeration types (one control and one test) specifying a representation different than the default representation given by the compiler for the test enumeration type, and allowing the control enumeration type to take the default representation. By performing identical operations on objects declared using the test and control types in a loop and measuring the difference in the two loops' execution times, the execution time associated with the use of the enumeration representation clause could be determined.

Potential compiler optimization of the test was controlled by a method used in the PIWG benchmarks, shown in an abbreviated form in figure 3. GLOBAL, A_ONE, and the procedure REMOTE are declared in package REMOTE_GLOBAL, whose body is compiled after that of the benchmark, thus the values of A_ONE and GLOBAL will now always be known to the compiler when the benchmark is compiled. Therefore, the sequence of statements in the benchmark cannot be changed by an optimizing compiler, because the effect of the optimization could alter the logical results of the code.

Methods for eliminating operating system interference from paging and execution of other processes have little to do with the design of the benchmark itself, but rather with the environment in which the test will be run. Benchmark tests should be run on a system with no other user processes in

```

package REMOTE_GLOBAL is -- for explicit control
    A_ONE : INTEGER;      -- of optimization
    GLOBAL : INTEGER := 1 ;
    procedure REMOTE;
end REMOTE_GLOBAL;

with REMOTE_GLOBAL; use REMOTE_GLOBAL;
procedure TEST is
begin
    for loop_counter in 1 .. 100 loop
        GLOBAL := GLOBAL + A_ONE;
        REMOTE;
        <Test feature statements>
    end loop;
end TEST;

package body REMOTE_GLOBAL is -- must be compiled last
    LOCAL : INTEGER; -- will be set to 0 at elaboration
    procedure REMOTE is
    begin
        GLOBAL := GLOBAL + LOCAL;
        exception
            when NUMERIC_ERROR => REMOTE ;
            -- cannot happen if test is working
    end REMOTE; -- ( prevents inlining )
begin
    A_ONE := 1 ;
    LOCAL := GLOBAL - A_ONE; -- really a zero but
end REMOTE_GLOBAL; -- compiler doesn't know

```

Figure 3. PIWG Optimization Control Package

concurrent execution and with as many system processes disabled as possible (Clapp and others, 1986: 764). I eliminated much of the operating system interference by keeping the size of the benchmark small (to reduce paging) and by running the benchmark when the host computer had a low load, to reduce the effect of other processes on the timing results from the benchmark.

Although the prototype test was to be run on a time-sharing system, I did not eliminate all operating system

interference. The goal of this thesis was to develop benchmarks for compilers targeted to embedded computers. Operating system interference is not a problem with these systems because the user has much greater control over the run time library (or operating system) and can ensure that only the benchmark is running when testing the compiler.

Repeatability of the tests was built into the benchmark by providing a sound performance data reporting capability and by making no assumptions about a set of conditions before the test started. Repeatability of the test was proven in its actual use.

The PIWG benchmarks and benchmark support software were of great use in the design for collection of timing data and for avoiding optimization of the test programs. A description of the PIWG software used in this thesis and instructions on how to obtain machine-readable copies may be found at Appendix C. Using the design philosophy of the PIWG benchmarks, I designed a performance benchmark that consisted of (1) a control loop containing an operation on an enumeration object without the use of an enumeration representation clause; and (2) a test loop containing an operation on an enumeration object that used an enumeration representation clause. This dual-loop approach, as described in Chapter 3, is superior to others because it isolates the feature and eliminates the time associated with overhead, such as loop control.

The design of this test, shown in figure 4, was modeled after test designs for other Ada language constructs in the PIWG suite of benchmarks (PIWG, 1986). In this design the

```
Control_Time_Start := SECONDS(CLOCK);
while <condition> loop
    <optimization control statements>
    <control version of feature>
end loop;
Control_Time_Stop := SECONDS(CLOCK);

Test_Time_Start := SECONDS(CLOCK);
while <condition> loop
    <optimization control statements>
    <test version of feature>
end loop;
Test_Time_Stop := SECONDS(CLOCK);

Difference_Time :=
    (Test_Time_Stop - Test_Time_Start) -
    (Control_Time_Stop - Control_Time_Start);
```

Figure 4. Performance Benchmark Structure.

current CPU time is retrieved from the SECONDS(CLOCK) function (DoD, 1983:Ch 9, 11) before execution of the control loop. The control loop will be executed a sufficient number of times to obtain the timing accuracy desired. Optimization control statements, as described earlier, are placed in the control loop, followed by statements that do not use the feature being studied. At the end of the control loop, the current time is again retrieved. The test loop is identical to the control loop except that the feature being studied is used in the test loop statements. The calculation at the end of the test removes all bias associated with the loop overhead and calls to the clock function, leaving the time associated with the use of

the feature. This calculation assumes that the test loop will take longer to execute than the control loop since use of the feature is normally assumed to add complexity to the test loop. It is possible that the default (or control) use of the features is *less* efficient than the test use of the feature. In such cases, one should determine the absolute value of the difference to determine the *increase* in performance attributable to the test version of the feature.

Problems in Prototype Test Design and Certification.

Recall that the Ada LRM states, "An implementation may limit its acceptance of representation clauses to those that can be handled simply by the underlying hardware" (DoD, 1983:Ch 13, 2). Therefore, although an implementation may implement a feature such as enumeration representation clauses, that implementation may restrict how the feature may be used. These restrictions, in turn, may cause some validation tests to be inapplicable for that implementation. If a test is ruled inapplicable, the test is not run when the compiler is validated. For example, an implementation for enumeration representation clauses may allow the clause for explicitly declared enumeration types, but not for derived types. Any validation tests for derived types would therefore be inapplicable, and the compiler simply would not be tested in that area.

The difficulty ruling certain tests inapplicable is that where a number of sub-objectives are listed for one test objective, separate test procedures must be written for each

sub-objective, because some of the sub-objectives may be inapplicable. For instance, in the example presented earlier, if a test procedure tested enumeration representation clauses for both explicitly declared and derived types, the test would be declared inapplicable, even though the compiler implemented enumeration representation clauses for explicitly declared types. Because compilers may provide no, partial, or full implementation of a feature, validation tests must be designed to test all possible implementations. This makes the number of tests that must be written increase dramatically as the number of sub-objectives of a validation test objective increases.

A seeming flaw in the application of this benchmarking approach is the assumption that use of a feature in an application will cause the execution time for a sequence of statements using the feature to increase. This may not be the case. In fact, preliminary test of the benchmarks for enumeration representation clauses showed the test loop took less time to execute than the control loop, resulting in a negative time difference between the loops. Other researchers have found similar anomalies using the dual-loop method (Altman, 1987:3).

The flaw is not in the dual-loop method, but in how it is applied. In the prototype benchmarks, I was unable to eliminate interference from the operating system, which distorted the benchmark's execution. When the dual-loop benchmarks were executed on embedded computers (with no

operating system interference) these timing anomalies were eliminated (Klemens, 1987).

Address Clauses and Interrupts Test Design

Now that the description of prototype test detailed design has been completed, let us proceed to describe the detailed design of the actual experimental tests.

Detailed Validation Test Design. As stated earlier, the design objectives were taken from the ACVC Implementers' Guide. The 23 test objectives test the ability of a compiler to detect illegal uses of the address clause and to permit legal uses of the clause. All developed tests should be passed by a compiler providing a full implementation of the address clause. Therefore, class A or class B test programs were written for all test objectives. The class A tests do not contain run-time checks for use of the address clauses, but I verified that these could easily be changed to class C (run-time) tests by checking that the addresses given for the objects, subprograms, etc. in the test are consistent with the addresses given by the 'ADDRESS attribute applied to the same objects, subprograms, etc. during the test execution (DoD, 1983:Ch13, 12). No other classes of tests could be reasonably applied to this feature, since it affects only the location of objects and executable statements when the test is compiled.

Detailed Performance Evaluation Test Design. The benchmarks for address clauses and interrupts were designed to determine the time it takes for an application using Ada's

interrupts feature to respond to an interrupt, because this time is critical to embedded applications (Clapp and others, 1986:771). The objective of these benchmarks was to measure the delay between the time the interrupt is raised and the time the interrupt handling routine is entered (Seward, 1987).

Recall from Chapter 3 that the interrupts feature of Ada allows an application to associate an interrupt with a task entry, so that, when the interrupt occurs, the task entry is called and the accept block for the entry, if any, is executed. The detailed design of these benchmarks was based on three variables in designs using this feature: (1) the number of interrupts that will be handled, (2) the number of tasks that will be used to handle these interrupts, and (3) the number of entries in each of the interrupt handling tasks.

The semantics of Ada tasking allow developers to write one task with several entry points, several tasks with one entry, or any combination in between to handle the 16 possible interrupts that MIL-STD-1750A computers may generate (DoD, 1982:19). Table II shows the configuration of the the benchmarks designed for this feature. In all of these tests only one interrupt is raised, that of a floating point overflow, generated by the benchmark itself. In order to isolate and study the interrupt delay time associated with one interrupt (i.e., floating point overflow), I designed the benchmarks so that only one interrupt could cause a call to a task entry. If task entries had been associated with other

Table II
Interrupts Benchmark Configuration.

Test Name	Interrupts Handled	Tasks	Entries per Task
INT_TEST1	10	10	1
INT_TEST2	10	5	2
INT_TEST3	10	2	5
INT_TEST4	10	1	10
INT_TEST5	3	3	1
INT_TEST6	3	1	3
INT_TEST7	1	1	1

interrupts that are known to occur during a program's execution, these other interrupts could have had an affect on the delay time of the interrupt being studied. For this reason, six of the MIL-STD-1750A's 16 interrupts were not associated with task entries in the benchmarks developed. Interrupt 5 (executive call), interrupt 7 (timer A interrupt), and interrupt 9 (timer B interrupt) were not handled because they would occur too frequently; interrupt 0 (power down), interrupt 1 (machine error), and interrupt 15 (spare) should not be handled by an application.

INT_TEST1, INT_TEST2, INT_TEST3, and INT_TEST4 were designed to determine the effect the number of tasks and entries per task had on interrupt response time. One would expect that variability in these results would come from the increased overhead incurred by the task scheduler for the benchmarks with a high number of tasks. On the other hand, the rendezvous logic for those tasks with single entry points is less complicated and should be more efficient.

INT_TEST5, and INT_TEST6 were designed to study the same effects but restricting the interrupts that were handled in the application to only those interrupts that were expected to occur during the execution of floating and fixed point instructions. Again, the effect of the number of tasks and entries per task on interrupt response time was examined.

Finally, INT_TEST7 was designed as a baseline against which the other benchmarks may be compared. It contains the minimal number of tasks, entries, and interrupts handled.

The benchmarks were all designed to raise the floating point overflow interrupt (interrupt 3) (DoD, 1982:19) by adding one floating point object whose value is just below the largest floating point number that may be stored to another floating point object whose value is such that the operation causes an overflow. No matter what the task/entry mix of the benchmark, at least one task entry will have an address clause that ties the entry to MIL-STD-1750A interrupt 3. The example shown in figure 5, an abbreviated version of INT_TEST7, gives the format for the interrupt delay benchmarks. A call is made to a procedure that will return the current CPU time before the floating point overflow is generated in the main procedure. The floating point overflow is then caused, which should effectively generate a task entry for the task entry tied to interrupt 3. The accept block for the entry is then entered, and a call is made to GET_ALL_TIMES to retrieve the CPU time. This time value is then stored in a global variable that is made visible to the task and the main procedure through the

```

package GLOBAL_INTERRUPT_MAKER is
    function FLOAT_RETURN return float;
end GLOBAL_INTERRUPT_MAKER;

with TIME_PACKAGE_1750A; use TIME_PACKAGE_1750A;
with GLOBAL_INTERRUPT_MAKER;
procedure INT_TEST7 is
    float_object    :FLOAT:= 0.500000 * 2.0 ** 127;
    delay           : TIME_1750A := 0.0;
    clock_bias_start : TIME_1750A := 0.0;
    clock_bias_stop  : TIME_1750A := 0.0;
    before_interrupt : TIME_1750A := 0.0;
    after_interrupt  : TIME_1750A := 0.0;
    pragma SHARED(after_interrupt);

    task INT_HANDLE_3 is
        entry interrupt3;
        for interrupt3 use at 3;
        pragma PRIORITY(3);
    end INT_HANDLE_3;
    task body INT_HANDLE_3 is
        begin
            loop
                accept interrupt3 do
                    GET_ALL_TIMES(after_interrupt);
                end interrupt3;
            end loop;
        end INT_HANDLE_3;
    begin -- INT_TEST7
        for i in 1 .. max_values loop
            GET_ALL_TIMES(clock_bias_start);
            GET_ALL_TIMES(clock_bias_stop);
            GET_ALL_TIMES(before_interrupt);
            float_object := float_object +
                GLOBAL_INTERRUPT_MAKER.FLOAT_RETURN;
            delay := after_interrupt - (before_interrupt +
                clock_bias_stop - clock_bias_start);
        end loop;
        abort INT_HANDLE_3;
    end INT_TEST7;

package body GLOBAL_INTERRUPT_MAKER is
    function FLOAT_RETURN return float is
        begin
            return 0.50000 * 2.0 ** 127;
        end FLOAT_RETURN;
    end GLOBAL_INTERRUPT_MAKER;

```

Figure 5. Example Interrupt Delay Benchmark

Ada pragma SHARED. After the accept block of the task entry has finished its execution, control returns to the loop in the main procedure and the interrupt delay time is determined. By subtracting the time for a call to the GET_ALL_TIMES routine, the effect of the timing calls may be eliminated. A sample of the benchmarks developed in this thesis and instructions on how to obtain machine-readable copies are at Appendix C.

Validation Test Design Problems. The primary difficulty with writing validation tests for address clauses is the representation of the address in the test. The syntax of the address clause is:

```
for simple_name use at simple_expression;  
(DoD, 1983:Ch 13, 7).
```

The *simple_expression* specifies the address for the entity given by *simple_name*, and the expression must be of the implementation-defined type ADDRESS, declared in the Ada Package SYSTEM (DoD, 1983:Ch 13, 7,10).

While an address given in the address clause, whether interpreted as an address or an interrupt level, will ultimately be translated to some physical memory address on the underlying hardware, the type ADDRESS may be the pre-defined Ada type integer, positive, natural, access, private, limited private, or even some other type declared by the implementation. Therefore, validation tests had to be designed in such a way that addresses were not explicitly given in the tests. Consider the simple example at figure 6 of an address clause. In this example, an object of type integer called

```
counter : Integer;  
for counter use at 100;
```

Figure 6. Address Clause Example

counter has been declared, and the compiler has been instructed to store *counter* at address 100. This is legal for compilers that recognize 100 as a valid value for type ADDRESS. Another compiler, with a correct implementation of the address clause, may reject this example because it has an additional restriction that addresses for objects be greater than some pre-defined number, for example, 1000. Because of differences in computer architecture and restrictions an implementation may place on address clauses, there could be a separate set of legality rules for address clauses for each Ada compiler. The problem, then, is how to write syntactically legal validation tests that all Ada compilers claiming a correct implementation of address clauses and interrupts could pass.

The first solution is to write tests specifically for each implementation. Currently there are 137 validated Ada compilers, and the number is growing (AdaIC, 1987). Given the number of Ada compilers, this would prove to be unworkable because it would severely complicate the maintenance of the ACVC test suite, as well as running counter to software engineering principles of usability and portability.

Another solution, and one that has been used in tests for other implementation defined language features in the ACVC, is

to replace the feature with a *macro* (Brashear, 1987a), a symbolic value that would be modified by each implementation. The example in figure 6 would be modified as shown in figure 7.

```
counter: Integer;  
for counter use at $object_address;
```

Figure 7. Address Clause Example with Macro

This 'master version' test would then be modified by each compiler implementer when the compiler was validated, replacing *\$object_address* with a legal address for his implementation. This approach, although workable, would be cumbersome, given the fact that these macros may appear hundreds of times in the tests for just one language feature.

The approach used to retrieve legal addresses in this thesis project, then, was to write a package ADDRESS_PACKAGE that declared a number of objects of type ADDRESS in the visible part of the package. A number of objects, subprograms, packages, tasks, and task entries were declared in declarative blocks in the executable region of the task body. The addresses for all these entities were extracted using the 'ADDRESS attribute (DoD, 1983:Ch 13, 12) and stored in the visible address objects. Since a declaration of an entity in a block is effective only for the duration of the block, it was legal to reuse these addresses because the entities declared in the ADDRESS_PACKAGE body no longer existed when the addresses were used in a validation test. This worked well with one

compiler that placed few restrictions on the *simple expression* in the address clause. Another compiler restricted the address clause to static values that must be set when the compilation unit in which the address clause is used is compiled. For this compiler, constant objects of type ADDRESS were declared and assigned values that would be accepted by the implementation. This method means that each implementer will have to provide the package body for ADDRESS_PACKAGE tailored to his implementation when running the validation test suite.

While this approach adds another package to the ACVC test support software, it eliminates the need for large number of changes to ACVC tests that would be required to tailor those tests to a particular implementation, and eliminates the need for tests written particularly for each implementation.

Another problem, discussed earlier with enumeration representation clauses, is that implementations may restrict the use of address clauses. This means that a separate test program must be written for each combination of sub-objectives of a test objective. For example, test objective 11 for address clauses states:

Check whether an address clause can be given for an object declared in a declarative part.

Implementation Guideline: Use a variable and constant having the following types: enumeration, integer, floating point, fixed point, array, record, access, private, limited private, and task.

Implementation Guideline: Include a check for declarative parts of subprograms, blocks, and package bodies (Goodenough, 1986:Ch 13, 54).

Since an implementation could conceivably restrict its acceptance of address clauses to those given for integer variables, 20 test programs would have to be written for this test objective, because there are 20 different combinations of objects, variables, and constants. Each of the sub-objectives would have to be written as a separate test, because any test that contained a use of the feature that the compiler restricted would be ruled *inapplicable* (Wilson, 1987a). Since each compiler's implementation may vary between none and complete, the tests must be written so that if a test is ruled inapplicable, it does not contain a test for a use of the feature that the implementation does support.

Performance Evaluation Test Design Problems. In the course of this research, I found that the implementation of package CALENDAR provided by the compiler being used to develop the benchmarks did not return the current CPU time with the precision necessary to accurately measure interrupt delay time. This compiler's implementation of the pre-defined type DURATION is:

```
type DURATION is delta 2.0 ** (- 12)
                    range -86_400.0 .. 86_400.0;
```

The current CPU time value returned by the SECONDS function in package CALENDAR is a subtype of DURATION, therefore the CPU time is accurate only to 244 microseconds. Past experience has shown other interrupt delay times to be in the 10 to 100 microsecond range, thus the precision of the elapsed CPU time in package CALENDAR was inadequate for this research. An

alternative package, TIME_PACKAGE_1750A was written that included three procedures: GET_ALL_TIMES, RESET_INT_VECTORS_7_AND_9, and GET_TIMERS; and declared a floating point type, TIME_1750A. GET_ALL_TIMES calls GET_TIMERS to retrieve the current value of the TIMER A register of the MIL-STD-1750A. Because the TIMER A register is incremented every 10 microseconds (DoD, 1982) GET_ALL_TIMES can calculate the current CPU time (in seconds) by dividing the value of the register by 10,000 / second. If, for example, the current value of the TIMER A register was 1234, the elapsed time since TIMER A was set to 0 would be 12,340 microseconds or 0.01234 seconds. GET_ALL_TIMES, then, was able to return the current CPU time accurate to 10 microseconds. After its correct operation was verified, the GET_ALL_TIMES procedure in TIME_PACKAGE_1750A was used to retrieve CPU time in the benchmarks in the remainder of this research. Appendix F contains a detailed description of TIME_PACKAGE_1750A.

Summary

This chapter described the detailed design of the validation tests and benchmarks developed for the prototype feature, enumeration representation clauses, and the operational features, address clauses and interrupts.

The validation tests were designed to test a compiler's ability to recognize legal uses of the feature and also to detect illegal uses of the feature. The problem of non-portability of the tests because of differing implementations

of the implementation-defined type ADDRESS was solved by placing all objects of type ADDRESS that would be used in the validation tests in one package and having the implementer supply the package body.

The benchmarks for interrupt delay time measure the delay from the time an interrupt is caused with a floating point overflow to the time the accept block of the interrupt handling task is entered. A support package, TIME_PACKAGE_1750A, was developed to return CPU time with greater precision than available with the implementation of package CALENDAR found in the compilers used in this research.

The next chapter will validate (or certify) the approach and the tests developed in this research, report the results of the tests, and analyze the results.

V. Analysis and Results

The purpose of this chapter is to certify the test software written in this research and to report the results of the validation and performance evaluation tests for the Chapter 13 features studied. I will use the term 'certification' instead of 'validation' to avoid confusion with the validation tests. The certification of these tests will ensure that the tests do indeed test what they claim to test.

In order to keep the potential distribution of this thesis as wide as possible, no actual compiler names will be used when reporting validation or benchmark results. The compilers that were used will be generically described as shown in table III below.

Table III
Generic Compiler Description

Compiler	Host	Target
A	DEC VAX-11/780*	DEC VAX-11/780
B	DEC VAX-11/780	MIL-STD-1750A
C	DEC VAX-11/780	MIL-STD-1750A

All of the compilers used in this thesis were validated when the research was performed. The benchmarks developed in this thesis were executed on a Sperry 1631 MIL-STD-1750A processor.

* VAX is a trademark of Digital Equipment Corporation.

Certification of Validation Tests

This section will certify that the validation tests written in this thesis will indicate whether a compiler's implementation of address clauses and interrupts conforms with the specification for those features in the LRM. Recall from Chapter 4 that two classes of validation tests were written: Class A tests, which determine a compiler's ability to recognize legal uses of a feature, and Class B tests, which determine a compiler's ability to recognize illegal uses of a feature. I argue that certification of these validation tests arises from the following evidence: the design of the tests was made as simple as possible; potential for error in the use of the address clause was minimized by localizing all address values in one package; all non-address clause syntax errors were eliminated; complex test cases were developed separately; implementation-dependent constructs were avoided; and finally, the tests were reviewed for correctness by the primary developers of tests for the ACVC test suite. Each of these contributions to test certification is described below.

Test Simplicity. To test whether a compiler could use a feature under a specific set of conditions, the validation tests were written to include those conditions and no others, lest spurious side-effects be introduced.

Minimization of Error-Prone Constructs. This technique was used to certify the development of the ADDRESS_PACKAGE for the address clause and interrupts tests. Each of these tests used a constant or variable from the visible part of

ADDRESS_PACKAGE for the address of a construct in the address clause. By declaring all addresses in one package and using that package for all tests, the potential for error in the syntax of the address clause was greatly reduced. Such use eliminated the need to enter explicit addresses in each of the many (110) tests, while ensuring that the Ada compiler would detect errors such as misspelled address variable names by identifying them as undeclared variables, since all variables must be explicitly declared in an Ada program.

Certification of Test Correctness Without the Feature. In developing the validation tests for address clauses and interrupts, I assumed that the Ada compiler being studied correctly implemented all other language constructs used in the test. As each test was developed for a specific use of Ada's address clause, a similar version that did not make use of the address clause (but identical in every other way) was also developed, compiled, and examined for syntax errors. Performing these parallel compilations ensured that the validation tests developed did not contain illegal uses of other Ada constructs that could affect the outcome of a validation test.

Combinations Tested Separately. The features being studied, address clauses and interrupts, often could be used in a number of different ways. Because compilers are free to reject the use of a feature under certain conditions (e.g., allowing address clauses for variables but not for constants), one test was written for each use. Writing separate tests for

combinations of the feature's use ensured that none of the tests would be ruled inapplicable while they still contained tests for uses of the feature that a compiler claimed to support. Therefore, a compiler would be tested for all uses of the feature that it claimed to implement. Should a compiler fail a test, the specific nature of the test would also help to pinpoint the set of circumstances that caused the compiler to fail.

Eliminate Implementation-Dependencies in Tests. In each test design I used only those constructs (excepting address clauses and interrupts) that all compilers *must* support. For example, all compilers have to support a data type named INTEGER but do not have to support the type named SHORT_INTEGER, so tests that required integer objects were declared using the INTEGER type.

Review by ACVC Test Developers. Finally, the tests developed in this thesis were reviewed for accuracy and correctness by the primary ACVC test developers for the Ada Validation Facility. These developers found that the validation tests developed in this research were "on target," the approach used in developing the tests was "sound," and the use of ADDRESS_PACKAGE to solve the problem of non-portability of address clauses agreed with the developers' solution for this problem (Brashear, 1987b).

The design techniques and software reviews built in to the design of these tests ensure that the test suite developed for address clauses and interrupts will detect illegal

implementations of these features and certify valid implementations of these features. The test simplicity, minimization of error-prone constructs, and parallel testing without the feature preclude the introduction of erroneous usage of Ada constructs that are not being tested.

By designing the test suite to test elementary uses of address clauses and interrupts, I have designed a test suite that may be applied to many compilers with varying implementations of these features. This testing of elementary uses of the feature avoids more complicated combinations of the feature's use, which makes the test more portable. The review by the primary ACVC test developers provides the assurance that a suite similar to that developed in this thesis would be used by the ACVC to perform actual Ada compiler evaluations.

Therefore, this suite may be used to answer the first question of the thesis problem. Developers of embedded applications, before they use a compiler's address clause or interrupt feature in their application, may apply these validation tests developed in this thesis to their compiler and know whether that compiler's implementation is valid, and possibly suitable for their application.

Validation Test Results

Because of the varying level of Chapter 13 implementations that the LRM allows, discussed in chapter 4 of this thesis, some validation tests for Chapter 13 features may be inapplicable for certain compilers. The only way for a

compiler to fail a validation is for it to fail a test that is applicable. The results for the validation tests will be reported as *passed* (the compiler implemented the feature correctly), *failed* (the compiler did not implement the feature correctly), or *inapplicable* (no test was made).

Enumeration Representation Clauses. The primary result obtained from this set of tests was a greater understanding of the thesis problem of validating Chapter 13 features. While compiler A passed all four tests that were written, these four tests examined only four of the 13 test objectives for enumeration representation clauses. No determination of a language feature's validity of implementation may be made on such incomplete testing, because a compiler must pass all applicable tests to be certified as a valid Ada compiler. In the case of compiler A, all test objectives were applicable to its implementation of enumeration representation clauses, so all tests would have to be run before it could be certified as a valid Ada compiler.

Address Clauses and Interrupts. One hundred and ten tests were written to test the 18 objectives for address clauses and the five objectives for interrupts. A complete list of the tests run on the compilers and of the results of those tests is provided at Appendix D.

Compiler A passed 58 tests, failed none, and was not tested on 52 because they were inapplicable. Compiler B passed 63 tests, failed 1, and was not tested on 46 inapplicable

tests. Compiler C passed 58 tests, failed none, and was not tested on 52 inapplicable tests.

The test that compiler B failed, test BD5102A, tests the ACVC Implementers' Guide objective: "Check that an address clause cannot be given for a (task) entry family" (Goodenough, 1986:56). This is a class B test that should fail compilation because it includes an illegal use of the address clause. Compiler B, however, compiled test BD5102A without detecting the error, thus failing the test. After discussing this failed test with the compiler developers, the error that caused compiler B to fail test BD5102A was corrected. Because the later version passed all tests, and I determined that the corrected version did not affect the benchmarks developed in this thesis, compiler B was used to develop and run the benchmarks for interrupt delay time in this thesis.

Using the guidelines of the Ada Validation Facility (Wilson, 1987a), compiler A and compiler C's implementations of address clauses and interrupts would be ruled valid because they passed all applicable tests. Compiler B's implementation originally would have been ruled invalid because it originally failed test BD5102A. When the compiler was corrected and all applicable tests were passed, then compiler B's implementation would be ruled valid as well.

Certification of Performance Evaluation Tests

This section will answer the question: how do we know that the benchmarks developed in this thesis for interrupt response

time present results (or even study the problems) of interest to the embedded applications developers? This question will be answered by reviewing how the interrupts feature is currently implemented and used, then expanding upon this review to identify how the feature could alternatively be used. I will then show that the benchmarks developed to measure interrupt delay time for interrupts associated with task entries measure the effects of the various uses of this feature.

Current Use of Interrupts Feature. Because compilers that implement address clauses for task entries are just recently becoming widely used, there currently does not exist a body of documented practical experience in the feature's use. I found, however, through personal and telephone interviews with Ada compiler and application researchers and developers that use of task entries associated with interrupts is currently being dictated by the compiler's implementation of the feature or by personal choice (Johnson, 1987). I found that some developers were using compilers that restricted the interrupt-handling task to be a single-entry task. Other developers were using a single-entry task to handle each interrupt, even though their compiler did not place this restriction on the feature's use. I did not find, however, that this design of interrupt handling tasks was based on an informed investigation of the efficiency of possible alternatives. These alternatives will be discussed below.

Alternative Use of Interrupts Feature. The semantics of Ada tasking present a number of alternative methods for

associating any number of interrupts with task entries using an address clause. One may wish to handle all of the possible interrupts in an application or just a subset (even of size one). Assume, for example, a developer wishes to handle four of 16 possible interrupts in his application. Although there are others, three possible designs for the tasks are: (1) declare one task with four entries, each associated with one of the interrupts; (2) declare two tasks with two entries per task; or (3) declare four tasks with one entry per task. The three major considerations, then, in the design of interrupt handling task(s), defined by the syntax of Ada and the configuration of the target, are: (1) the number of interrupts to be handled; (2) the number of tasks to handle the interrupts; and (3) the number of entries per task.

The benchmarks developed in this thesis form a representative sample of the various techniques for associating tasks with interrupt. Recall from Table 1 in chapter 4 that INT_TEST1, INT_TEST2, INT_TEST3, and INT_TEST4 are all designed to handle a high number of interrupts (10) on the MIL-STD-1750A's 16 possible interrupts (DoD, 1982:19), with designs ranging from 10 tasks with one entry each to one task with 10 entries. These benchmarks produce a comparative measure of interrupt delay time for the interrupts feature as it is currently being used, and as it could be used. INT_TEST5 and INT_TEST6 test the task entry mix with a medium (three) number of interrupts being handled. INT_TEST7 provides the baseline, handling one interrupt with one single-entry task.

By measuring the interrupt delay time for this feature as it is currently being used as well as for other possible uses, the benchmarks provide the answer to the question: "How efficient is this feature's (address clauses for interrupts) implementation?"

The reader might ask, "Does it make sense to write one task to handle a number of interrupts?" The answer is "yes," because interrupts are often related, and the interrupt-handling functions are similar. For example, if the operations one might perform to handle a floating point overflow and a fixed point overflow are similar, it would be logical to localize these operations in one interrupt handling block.

The results from these seven benchmarks demonstrate how they provide comparative information on the efficiency of the compiler. Not only do they indicate whether the interrupts feature is efficient enough for a particular application, they also indicate the most efficient way to use this feature.

Some developers were not using Ada's interrupts feature because they said it was more efficient to handle interrupts in the run-time executive or operating system. Others were using the feature in only one way (one single-entry task for each interrupt being handled) and had not investigated other approaches. If the embedded systems developer is to write an application that will execute as efficiently as possible, he must know the most efficient way to use a particular feature for his compiler/target combination. The benchmarks for interrupt delay time provide these measures of efficiency for

the interrupts feature as it may be used in an application. The benchmarks will either provide additional support for the single task/multiple entry method of interrupt handling in an application or prove it lacking and provide the embedded applications developers the impetus to use alternative, more efficient, designs for interrupt handling.

Performance Evaluation Test Results

This section will report the results of the benchmarks developed in this thesis. The benchmarks were compiled with only compiler B and run on the Sperry 1631 MIL-STD-1750A computer because B was the only compiler available during my research that implemented the interrupts feature. The results reported here, therefore, will not be used to compare the relative efficiency of one compiler to another. The results do indicate that these benchmarks may be applied to compilers supporting address clauses for interrupts to determine the most efficient use of the feature.

Enumeration Representation Clauses. Test A13_3_3C, designed to measure the effect of performing a successor operation on an enumeration object given an alternate representation, was run on one self-targeting compiler, compiler A. The result from this benchmark was a negative mean (test loop time - control loop time) and a large standard deviation. These results, skewed by the effect of the operating system on the benchmark, reinforced the need to avoid these distortions as discussed in Chapter 3.

Address Clauses and Interrupts. The seven benchmarks for measurement of interrupt delay time for interrupts associated with task entries determined the mean interrupt delay time of the floating point overflow interrupt. This calculation was based on 100 floating point overflows generated in a loop in the benchmark. Additionally, the total elapsed time for the benchmark is given as an indication of the overhead associated with the various methods of interrupt handling. The results of the benchmarks are presented in Table IV below. The mean and

Table IV
Interrupts Benchmark Results

Test Name	Mean (seconds)	Standard Deviation (seconds)	Elapsed Time (seconds)
INT_TEST1	7.365 E-4	7.441 E-6	3.819 E-1
INT_TEST2	7.731 E-4	7.020 E-6	4.497 E-1
INT_TEST3	7.958 E-4	7.691 E-6	5.240 E-1
INT_TEST4	7.906 E-4	5.914 E-6	6.360 E-1
INT_TEST5	7.395 E-4	7.641 E-6	3.799 E-1
INT_TEST6	7.585 E-4	6.314 E-6	4.708 E-1
INT_TEST7	7.341 E-4	6.965 E-6	3.810 E-1

standard deviation were calculated using a sample of 100 interrupts generated in each benchmark. The sample size of 100 was chosen to ensure a level of reliability in the results, so that a difference in the interrupt delay times could be demonstrated if it existed. The mean, standard deviation, and elapsed time between the first and last floating point overflow were calculated in the benchmark.

Assuming that the interrupt delay times are normally distributed, I used a two-sample t test and a level of significance of 0.01 to find that the mean interrupt delay time for INT_TEST1 was less than that for INT_TEST2, INT_TEST3, or INT_TEST4 (Larsen and Marx, 1986:364). Likewise, I found that the mean interrupt delay time for INT_TEST5 was less than that for INT_TEST6. The calculations for the two-sample t test are found in Appendix G.

These results show that the design of the interrupt handling tasks does have an effect on the interrupt delay time. Whether the benchmark is handling a high (10) or medium (3) number of interrupts, the most efficient approach in terms of minimum interrupt delay time and tasking overhead (as measured by the elapsed time of the benchmark) is to write a single entry task for each interrupt being handled in the application. This does not mean, however, that the single-entry task approach will be the most efficient for all compiler/target combinations. These benchmarks may be applied to other compilers to determine the most efficient approach for particular applications.

Summary

This chapter presented a certification of the validation and performance evaluation tests developed in this thesis, and reported the results of those tests on three validated Ada compilers. Validation test design techniques and a design review were described to certify that the validation tests will

determine the validity of a compiler's implementation of Ada's address clauses and interrupts features. The benchmarks were certified by showing that they measured interrupt delay time for the interrupts feature as it is currently used and as it could be used.

The results of the validation tests for three compilers, in which one compiler initially failed (and then later passed) one test, were detailed. This reinforced the idea that a *validated* Ada compiler may not implement all of the untested features correctly. Finally, the results of the benchmarks compiled with compiler B and executed on a Sperry 1631 MIL-STD-1750A processor were presented, showing the single-entry task approach for the design of interrupt handling tasks to be the most efficient approach for the compiler/target studied.

The next chapter reviews how well the work in this thesis solved the thesis problem, draws conclusions about this research, and lists recommendations for future study.

VI. Conclusions and Recommendations

This chapter summarizes how the approach used and software developed in this research answered the questions posed in the thesis problem, presents conclusions about this research, and outlines areas for future research.

The Problem Revisited

Recall from Chapter 1 that there were two questions to be answered by potential users of Chapter 13 features before these features were used in embedded applications: (1) does the compiler's implementation of the feature conform to the definition of that feature in the LRM and (2) is the implementation of the feature efficient enough for the users' application? The first half of the thesis problem, concerning the validity of the implementation of address clauses and interrupts was solved by the validation tests written for those features. The design techniques used in the development of these tests, as discussed in Chapter 5, ensure a complete, accurate, and error-free test suite for Address Clauses and Interrupts.

The second part of the thesis problem, the efficiency of the feature, was more difficult to define and quantify so that it could be tested. At the outset, the two metrics used most often to quantify 'efficiency' were time (execution speed) and space (amount of memory used). As the research progressed, time became the primary metric because of advances in the memory capability of the MIL-STD-1750A processor.

I was able to determine, through numerous interviews, how the interrupts feature was being used by embedded systems applications researchers and developers. From this I designed a suite of benchmarks and support software that measured the efficiency of the feature as it was currently used as well as alternative uses allowed by the syntax of the features specified in the LRM. These benchmarks, then, answer the second half of the question in the thesis problem.

Therefore, this thesis shows that the problem can be solved and demonstrates this fact by answering the validity and efficiency questions for address clauses and interrupts.

Conclusions

At the beginning of this thesis project, I felt that the emphasis of the research should be on benchmarking Chapter 13 features, and that validation of those features was a necessary (but not very difficult) step on the way to that goal. As the research progressed, it became increasingly apparent that validation and performance evaluation were complementary disciplines of equal importance. Before Ada researchers measure a compiler's efficiency with benchmarks, they *must* know that the feature or construct being studied is correctly implemented. If compiler validation is not performed, non-valid implementations may appear to be more efficient when benchmarked because these implementations may not provide all the capability of a valid implementation.

The approach used in this thesis project to determine the validity and efficiency of compilers' implementations of address clauses and interrupts may be applied to other features. This application assumes, however, that researchers and developers can identify the features to be tested and how to test them. The difficulty of this research is not so much in the design of validation tests and benchmarks, but in determining what to test, why to test it, and how to write a valid test. I was able to prioritize the Chapter 13 features for study by interviewing Ada researchers and developers to determine what features were most important to their applications. I mistakenly assumed that there would be a strong consensus among embedded application developers concerning what was 'most important' in Chapter 13. The reality was (and is) that compiler implementations are being improved just as new embedded applications problems are being solved. No one has a list of exactly what the 'problems of embedded applications' are. Although there is some overlap, each developer has his own set of problems, often unique to his application.

Just as there are varying levels of Ada's use in developing embedded applications, so too are there varying levels of interest in the Chapter 13 features. Some developers interviewed in this research said that they did not plan to use any Chapter 13 features, because they felt that the same functions could be performed more efficiently with assembly language, JOVIAL, or other languages for embedded

applications. DoD Directive 3405.1, which mandates the use of Ada in new applications, should increase the interest in how to test the validity and efficiency of Chapter 13 features. Embedded applications developers will either use Ada or request a waiver because the current technology cannot solve their problem (DoD, 1987). In either case, these developers will have to find objective answers to questions concerning the validity and efficiency of Ada compilers, with emphasis on the Chapter 13 features that are so vital to the success of embedded applications. Whether Ada is or is not used for developing new embedded applications, the decision will have to be an informed one.

Validation and performance evaluation tests also generate a positive result that is not part of their design, but is a by-product of their existence. This result is the cause-effect relationship one researcher found between the release of compiler benchmarks for Ada language constructs and efficient, complete implementations of those constructs in compilers released after the wide distribution of those benchmarks. This heightened cross-flow of information concerning efficiency, validity, and shortcomings of various compiler implementations will increase the pool of knowledge about those implementations, increase competition between compiler vendors, and result in improved development tools for those using Ada for embedded applications.

There has been some discussion that the distribution of the Ada Compiler Evaluation Capability (ACEC), described in

Chapter 2, may be limited because of concerns about misuse of the ACEC to label compilers as 'good' or 'bad' with respect to one construct or feature. Although the misuse of any evaluation tool is possible, the potential for increase in the quality of implementations of Ada compilers, I feel, outweighs the need to prevent misuse of the ACEC test suite. The AJPO should release the ACEC to widest possible distribution, perhaps placing it on the Ada Software Repository, in a fashion similar to that used for the SIGAda PIWG benchmarks, and encourage its proper use with an aggressive education program.

One of the goals of this thesis project was to develop benchmarks that could be applied to any Ada compiler targeted to the MIL-STD-1750A processor. As the research progressed, it became apparent that the implementation-defined features of each compiler used in these benchmarks made it difficult to run the benchmarks on another compiler/target combination. Just because an embedded application is developed in Ada does not necessarily mean that the application is portable without modification. Because certain features and constructs used in the application must be tailored to the underlying hardware, some redesign and modification may be necessary to re-host the application on another processor. The software engineering concepts that Ada enforces, (i.e. data typing, constraint checking, etc.) will make this process easier than it would be with an application developed in assembly language, because only the Chapter 13 features will have to be modified for the application to be run on the new processor.

Recommendations for Future Research

Although Ada compilers currently do not have to implement all of the features in Chapter 13, a complete validation test suite for those features would further stimulate development of implementations of Chapter 13 features. The goal of a future research project would be to complete the ACVC test suite for Chapter 13 of the LRM. The current test design objectives of the ACVC Implementers' Guide may have to be extended to allow the tests to check memory locations, word length, data alignment, etc., using tools other than test software written in Ada. Having the compiler check one feature, such as an address clause, by using another feature, such as the 'ADDRESS attribute, is not the best way to validate compiler features. If a compiler vendor is going to do a poor job implementing a feature, or do something that is not expected or illegal, one would assume that he would be smart enough to avoid inconsistency in his implementation by implementing a complementary construct (in this case the 'ADDRESS attribute) in a similar fashion. For example, if one uses an address clause to store an object at location 500, but the implementation actually stores the object somewhere else, the compiler implementer may implement the 'ADDRESS attribute to return 500 when it is applied to the object in question, indicating that the implementation of address clauses was valid. This future research would develop alternatives to using one compiler construct to determine the validity of another.

Another research topic, determining the areas of weakness of the ACVC, may require the efforts of many researchers. Some have argued that by allowing compiler vendors to know the exact structure of the ACVC test suite, minimal implementations may be designed to pass the specific tests. This future research project would seek out the holes in the ACVC, and develop tests to fill those holes. The project would determine whether or not an implementation that does not comply with all of the specifications in the LRM could possibly pass the validation test suite. In short, it would validate the validation test suite. Some may argue that developing validation tests for a validation test suite could be recursively applied indefinitely. Although a validation review of the ACVC test suite could not be proven complete and valid itself, it has the potential to uncover and correct errors and incompleteness in the ACVC test suite. The importance of the ACVC test suite to the development of Ada compiler technology makes this project well worth the effort.

Appendix A: Validation Test Objectives

This appendix contains the test objectives for the Chapter 13 validation tests developed in this thesis. All of the information in this Appendix is taken from the ACVC Implementers' Guide (Goodenough, 1986:Ch 13, 39-40, 53-56).

Enumeration Representation Clause Test Objectives

T1. Check that a record representation clause cannot be given for:

- an expanded name that denotes a record type;
- a name declared by an object declaration;
- a name declared by a subtype declaration;
- a type declared by a private type declaration prior to the full declaration of the type;
- an incomplete type prior to the full declaration of the type;
- a type having a subcomponent of an incompletely declared private type, prior to the complete declaration of the composite type;
- a type that is not an enumeration type

Implementation Guideline: In each case, use a clause that, if possible, would be allowed for the actual type or for the completely declared type.

T2. Check that two enumeration representation clauses cannot be given for the same type.

Implementation Guideline: The two clauses should specify identical representations.

T3. Check that an enumeration representation clause cannot be given:

- in a package specification for a type declared in an inner package specification;
- in a package or task specification, for a type declared in an enclosing package specification or declarative part;
- in a package body for a type declared in the corresponding package specification;
- after the occurrence of a body in a declarative part.

T4. Check that an enumeration representation clause cannot be given after a forcing occurrence for the type.

T5. Check that if an enumeration representation clause can be given, it can be given after an occurrence of the type name in an expression of a pragma.

T6. Check that the name of the enumeration type (or a subtype of the enumeration type) cannot appear as a choice in the aggregate or in one of the expressions.

Implementation Guideline: The name should be used in an attribute (e.g., 'VAL) that delivers a value of the required type.

T11. Check that an enumeration representation clause cannot be given for a derived enumeration type if the derived type definition imposes a constraint of if the parent type has derivable subprograms.

Implementation Guideline: Write separate tests for these two cases.

T12. Check that integer codes must be given for each enumeration literal of the type.

Implementation Guideline: Check that neither too many nor too few codes can be given.

Check that nonstatic integer codes are not allowed.

Implementation Guideline: Use nonstatic *universal_integer* expressions.

Check that a choice cannot be nonstatic.

T13. Check that the same integer code cannot be given for two enumeration literals.

Check that the integer codes must obey the predefined ordering relation for the type.

Implementation Guideline: Include some aggregates in which choices do not appear in the order defined for the type, and for which an ordering operator has been explicitly declared.

Check that a choice in the aggregate must be a value of the enumeration type.

T14. Check whether an enumeration representation clause can be given for an enumeration type. If so, check that such types can be used correctly in ordering relations, in indexing arrays, in attributes, and in generic instantiations.

Implementation Guideline: Include cases where the integer codes have negative values and in which they do not have consecutive values.

Implementation Guideline: Combine this check with various forms of aggregate: all choices named (when some enumeration literals are character literals); no choices named.

Check that an enumeration representation clause can be given in the visible or private part of a package for a type declared in the visible part.

Implementation Guideline: Repeat the checks for enumeration representation clauses given in a generic unit.

T15. Repeat T14 for a derived enumeration type, including when the parent type has an enumeration representation clause given.

T21. Check that the aggregate in an enumeration representation clause cannot be considered ambiguous.

Implementation Guideline: Declare more than one one-dimensional array type that has the enumeration type as its index subtype.

T22. Check whether an enumeration representation clause can be given for a type derived from a type declared in a generic instantiation.

Address Clause Test Objectives

T1. Check that the expression in an address clause must have the type SYSTEM.ADDRESS.

T2. Check that an address clause is illegal if a with clause naming the predefined package SYSTEM does not apply to the unit containing the address clause.

Implementation Guideline: Check for objects, subprograms, packages, tasks, and entries.

T3. Check that if an address clause is allowed, a with clause naming SYSTEM need not be given for the compilation unit containing the address clause as long as such a clause applies to the unit.

Implementation Guideline: Check for address clauses in package bodies, subprogram bodies, and subunits. Include a check for generic unit bodies and subunits.

T4. Check that an address clause cannot be given for a named number, an exception, a formal parameter of a subprogram, entry, or generic unit, a generic formal object, a generic subprogram, a generic package, a loop parameter, an object designated by an access value, a slice, or a component of an object.

Check that an address clause cannot be given for a library unit or a generic unit.

T5. Check that an address clause cannot be given for an expanded name or for a name declared by a renaming declaration.

Implementation Guideline: Include renamings of objects, subprograms, packages, tasks, and entries.

T6. Check that an address clause cannot be given:

- in a package specification for an object, a package, etc., declared in an inner package specification;
- in a package or task specification, for an object,

- a package, etc., declared in an enclosing package specification or a declarative part;
 - in a package body for an object, package, etc. declared in the corresponding package specification;
 - after the occurrence of a body in a declarative part.
- Implementation Guideline:* In particular, check for a subprogram body that also acts as the declaration of the subprogram.

T7. Check that an address clause cannot be given for subprogram if more than one subprogram with the same name is declared explicitly in the same package specification or declarative part.

Implementation Guideline: Include a generic instantiation and a renaming declaration as well as a subprogram declaration. Check that if an address clause can be given for a subprogram, it can be given when the subprogram is overloaded by:

- a subprogram declared in an outer declarative region or library package,
- an entry declaration (when the subprogram is declared in the task body),
- an implicitly declared derived subprogram.

T8. Check that the expression in an address clause must be a simple expression.

T11. Check whether an address clause can be given for an object declared in a declarative part.

Implementation Guideline: Use a variable and constant having the following types: enumeration, integer, floating point, fixed point, array, record, access, private, limited private, and task.

Implementation Guideline: Include a check for declarative parts of subprograms, blocks, and package bodies.

T12. Repeat T11 for generic units.

T13. Check whether an address clause can be given for an object declared in a package specification.

Implementation Guideline: Use a variable and a constant having the following type: enumeration, integer, floating point, fixed point, array, record, access, private, limited private, and task.

Implementation Guideline: Include a check that the address clause can be given in the private part for an object declared in the visible part.

T14. Repeat T13 for generic packages.

T21. Check whether an address clause can be given for a subprogram declared in a declarative part by a subprogram declaration or a generic instantiation.

Implementation Guideline: Check for a declarative part of a block, a package body, a subprogram, and a task body.

T22. Check whether an address clause can be given for a subprogram declared in a package a subprogram declaration or a generic instantiation.

Implementation Guideline: Include a check that the clause can be given in the private part for a subprogram declared in the visible part.

T31. Check whether an address clause can be given for a package declared in a declarative part by a package declaration or a generic instantiation.

Implementation Guideline: Check for a declarative part of a block, package body, subprogram, and task body.

T32. Check whether an address clause can be given for a package declared in a package by a package declaration or generic instantiation.

Implementation Guideline: Include a check that the clause can be given on the private part for package declared in the visible part.

T41. Check whether an address clause can be given for a task type or a single task declared in a declarative part.

Implementation Guideline: Check for a declarative part of a block, a package body, a subprogram, and a task body.

T42. Check whether an address clause can be given for a task type or a single task declared in a package

Implementation Guideline: Include a check that the clause can be given in the private part for a task or a task type declared in the visible part.

Interrupts Test Objectives

T1. Check that an address clause cannot be specified for an entry that has a parameter of mode out or mode in out.

T2. Check that an address clause cannot be specified for an entry family.

T3. Check that the name in an address clause for an entry cannot be an expanded name.

T4. Check that an address clause for an entry cannot be given within the declarative part of the task body.

T11. Check that if an implementation supports address clauses for entries, such a clause can be given for an entry of a task type as well as for an entry of a single task.

Appendix B: Validation Test Software

This Appendix contains a sample of the validation tests developed in this thesis. A machine readable version of the software is available from the Air Force Institute of Technology, Department of Mathematics and Computer Science (ENC), WPAFB OH 45433.

Validation Test Naming Convention

The name associated with the validation tests conforms to the naming convention for the tests in the ACVC test suite. The test names will be of the form NAME.ADA, with NAME containing up to nine characters defined below:

<u>Character Position</u>	<u>Description</u>
1	Class of test (A,B,C,D,E,L)
2	AIG chapter number (Hex)
3	AIG section number (Hex)
4	AIG subsection number or letter
5,6	AIG test objective number
7	Test sequence letter (A-Z)
8	Compilation sequence digit (0-9) (Not required)
9	'M' indicates main program (for several compilation units) (Wilson, 1987b).

By my using this convention, the tests are more likely to be understood and accepted by those familiar with the conventions of the ACVC test suite. This convention thus helps to satisfy the requirement that the tests clearly identify their objectives. The name of the test does this by referring to the test objectives in the ACVC Implementers' Guide (Goodenough, 1986) and Appendix A to this thesis.

Enumeration Representation Clauses Test Names

BD3001A.ADA
BD3002A.ADA
BD3004A.ADA
BD3012A.ADA

Enumeration Representation Clauses Sample Test:

```
-----
-- BD3012A
-- Check that integer codes must be given for each enumeration
-- literal of the type
--
-- Check that nonstatic integer codes are not allowed (in
-- giving integer codes to each enumeration literal of the
-- type)
--
-- Author: Capt Dan Joyce
-- Version: 1.1
-- Date: 1 Jun 87
-----

procedure BD3012A(one :string;
                  two:  string;
                  three: string;
                  four:  string;
                  five:  string) is

    type enum_type is (a1,a2,a3,a4,a5);
    type enum_type1 is (a1,a2,a3,a4,a5);
    type enum_type2 is (a1,a2,a3,a4,a5);

    for enum_type use (1,2,3,4);      -- Illegal, too few
                                      -- integer codes

    for enum_type1 use (1,2,3,4,5,6); -- Illegal, too many
                                      -- integer codes

    for enum_type2 use (one'length,two'length,three'length,
                        four'length,five'length);
                                      -- Illegal use of nonstatic
                                      -- Universal integer for Choices

begin
    null;
end BD3012A;
```

Address Clause and Interrupts Test Names:

```
BD5001A.ADA

BD5002A.ADA   BD5002B.ADA   BD5002C.ADA   BD5002D.ADA
BD5002E.ADA

AD5003A.ADA   AD5003B.ADA   AD5003C.ADA   AD5003D.ADA
AD5003E.ADA

BD5004A.ADA   BD5004B.ADA   BD5004C.ADA
```

BD5005A.ADA	BD5005B.ADA		
BD5006A.ADA	BD5006B.ADA	BD5006C.ADA	BD5006D.ADA
BD5006E.ADA			
AD5007A.ADA	AD5007B.ADA	AD5007C.ADA	BD5007A.ADA
BD5008A.ADA			
AD5011A.ADA	AD5011B.ADA	AD5011C.ADA	AD5011D.ADA
AD5011E.ADA	AD5011F.ADA	AD5011G.ADA	AD5011H.ADA
AD5011I.ADA	AD5011J.ADA	AD5011K.ADA	AD5011L.ADA
AD5011M.ADA	AD5011N.ADA	AD5011O.ADA	AD5011P.ADA
AD5011Q.ADA	AD5011R.ADA	AD5011S.ADA	AD5011T.ADA
AD5012A.ADA	AD5012B.ADA	AD5012C.ADA	AD5012D.ADA
AD5012E.ADA	AD5012F.ADA	AD5012G.ADA	AD5012H.ADA
AD5012I.ADA	AD5012J.ADA	AD5012K.ADA	AD5012L.ADA
AD5012M.ADA	AD5012N.ADA	AD5012O.ADA	AD5012P.ADA
AD5012Q.ADA	AD5012R.ADA	AD5012S.ADA	AD5012T.ADA
AD5013A.ADA	AD5013B.ADA	AD5013C.ADA	AD5013D.ADA
AD5013E.ADA	AD5013F.ADA	AD5013G.ADA	AD5013H.ADA
AD5013I.ADA	AD5013J.ADA	AD5013K.ADA	AD5013L.ADA
AD5013M.ADA	AD5013N.ADA	AD5013O.ADA	AD5013P.ADA
AD5013Q.ADA	AD5013R.ADA	AD5013S.ADA	AD5013T.ADA
AD5014A.ADA	AD5014B.ADA	AD5014C.ADA	AD5014D.ADA
AD5014E.ADA	AD5014F.ADA	AD5014G.ADA	AD5014H.ADA
AD5014I.ADA	AD5014J.ADA	AD5014K.ADA	AD5014L.ADA
AD5014M.ADA	AD5014N.ADA	AD5014O.ADA	AD5014P.ADA
AD5014Q.ADA	AD5014R.ADA	AD5014S.ADA	AD5014T.ADA
AD5021A.ADA	AD5022A.ADA		
AD5031A.ADA	AD5032A.ADA		
AD5041A.ADA	AD5042A.ADA		
BD5101A.ADA	BD5101B.ADA		
BD5102A.ADA			
BD5103A.ADA			
BD5104A.ADA			
AD5111A.ADA			

Address Clause Sample Tests

```
-----  
-- BD5001A  
-- Check that the expression in an address clause must have  
-- the type SYSTEM.ADDRESS  
-- Check illegal address given to an object  
--  
-- Date:      27 July 87  
-- Version:   1.2  
-- Author:    Capt Dan Joyce  
-----
```

with system;

procedure BD5001A is

```
    subtype bad_address_type is integer;  
    bad_address1 : bad_address_type := 0;  
    bad_address2 : positive := 1;  
    bad_address3 : integer := 2;  
    bad_address4 : natural := 3;  
    bad_address5 : float := 0.1;
```

```
    object1 : integer;  
    for object1 use at bad_address1;  
        -- Error. Address must be  
        -- of type SYSTEM.ADDRESS
```

```
    object2 : integer;  
    for object2 use at bad_address2;  
        -- Error. Address must be  
        -- of type SYSTEM.ADDRESS
```

```
    object3 : integer;  
    for object3 use at bad_address3;  
        -- Error. Address must be  
        -- of type SYSTEM.ADDRESS
```

```
    object4 : integer;  
    for object4 use at bad_address4;  
        -- Error. Address must be  
        -- of type SYSTEM.ADDRESS
```

```
    object5 : integer;  
    for object5 use at bad_address5;  
        -- Error. Address must be  
        -- of type SYSTEM.ADDRESS
```

```
begin  
    null;  
end BD5001A;
```



```

-----
-- AD5011A
-- Check whether an address clause can be given for an object
-- declared in a declarative part.
-- This checks for enumeration type variables declared in the
-- declarative parts of subprograms, blocks, and package
-- bodies.
--

```

```

-- Date:      24 July 87
-- Version:    1.4
-- Author:     Capt Dan Joyce
-----

```

```

with SYSTEM;
with ADDRESS_PACKAGE; use ADDRESS_PACKAGE;
with REPORT; use REPORT;
procedure AD5011A is

```

```

begin

```

```

TEST('AD5011A ', 'Check whether an address clause can be given ' &
    'for enumeration type variables declared in the ' &
    'declarative parts of subprograms, blocks and package ' &
    'bodies. ');

```

```

BLOCK1: declare

```

```

-- This tests the declarative part of subprograms
procedure procl is

```

```

    type enum_type    is (red, blue, green);
    enum_obj1:         enum_type;
    for enum_obj1 use at object_address1;
    begin -- procl
        null;
    end procl;

```

```

begin -- BLOCK1
    null;
end BLOCK1;

```

```

BLOCK2: declare

```

```

-- This tests the declarative part of blocks
    type enum_type    is (red, blue, green);

    enum_obj1:         enum_type;
    for enum_obj1 use at object_address2;

```

```

begin -- BLOCK2
    null;
end BLOCK2;

BLOCK3: declare

    -- This tests the declarative part of package bodies
    package PKG is
    end PKG;

    package body PKG is
        type enum_type is (red, blue, green);
        enum_obj1: enum_type;
        for enum_obj1 use at object_address3;
    end PKG;

begin -- BLOCK3
    null;
end BLOCK3;

RESULT;

end AD5011A;

```

Interrupts Sample Tests

```

-----
-- BD5102A
-- Check that an address clause cannot be specified for
-- an entry family.
--
-- Date:      8 Sep 87
-- Version:   1.2
-- Author:    Capt Dan Joyce
-----

with system;
with ADDRESS_PACKAGE; use ADDRESS_PACKAGE;
package BD5102A is

    type interrupt_level is range 0 .. 2;

    task task1 is
        entry family_entry1 (interrupt_level);
        for family_entry1 use at entry_address5;
            -- Illegal. Can't give
            -- address clause !
            -- family entry

    end task1;
end BD5102A;

```

AD-A189 554

VALIDATING AND EVALUATING ADA'S (TRADE MARK)
REPRESENTATION CLAUSES AND I (U) AIR FORCE INST OF
TECH WRIGHT-PATTERSON AFB OH SCHOOL OF ENGI

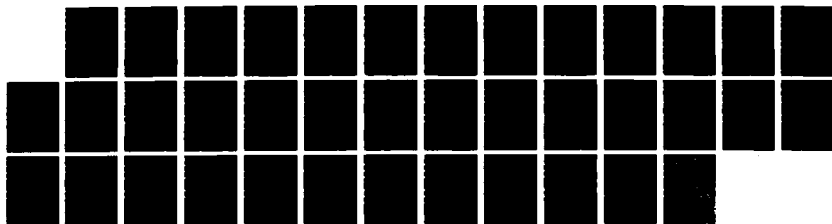
272

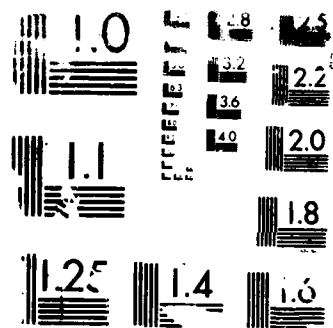
UNCLASSIFIED

D O JOYCE DEC 87 AFIT/GCS/NA/87D-2

F/G 12/5

NL





RESOLUTION TEST CHART

```
package body BD5102A is
```

```
    task body task1 is
    begin
        loop
            select
                accept family_entry1(0);
            or
                accept family_entry1(1);
            or
                accept family_entry1(2);
            end select;
        end loop;
    end task1;
end BD5102A;
```

```
-----
-- A5111A
-- Check that if an implementation supports address clauses
-- for entries, such a clause can be given for an entry of a
-- task type as well as for an entry of a single task.
--
-- Date:      9 Sep 87
-- Version:   1.5
-- Author:    Capt Dan Joyce
-----
```

```
with system;
with ADDRESS_PACKAGE; use ADDRESS_PACKAGE;
package AD5111A_PKG is
```

```
    task type task_type1 is
        entry entry1;
        for entry1 use at entry_address5;
    end task_type1;
-- Legal. Should be accepted
-- if implementation supports
-- entry addresses.
```

```
    task_object1 : task_type1;
```

```
    task task2 is
        entry entry2;
        for entry2 use at entry_address6;
    end task2;
-- Legal. Should be accepted
-- if implementation supports
-- entry addresses.
```

```
end AD5111A_PKG;
```

```

package body AD5111A_PKG is
  task body task_type1 is
    begin
      accept entry1;
    end task_type1;

    task body task2 is
      begin
        accept entry2;
      end task2;
    end AD5111A_PKG;

with AD5111A_PKG; use AD5111A_PKG;
with REPORT; use REPORT;
procedure AD5111A is
  begin
    -- The next two task calls allow the tasks to terminate
    -- and therefore the main task AD5111A can terminate

    task2.entry2;
    task_object1.entry1;

    TEST('AD5111A ', 'Check that an address clause can be given '&
      'entries for a task type as well as a single task ');
    RESULT;
  end AD5111A;

```

Appendix C: Performance Evaluation Test Software

I used the following PIWG programs to support the benchmarks developed for Chapter 13 features: A000001, containing the package DURATION_IO for reporting timing results; A000012, containing the CPU_TIME_CLOCK for DEC VAX computers; A000021 and A000022, containing the specification and body of the REMOTE_GLOBAL optimization control package. I also modified the PIWG package A000050 and the procedures A000052, A000053, A000054, and A000055 to return CPU times from in the format I needed in this benchmark. The source for this package and these modified procedures is included in this appendix.

The machine readable code for these modules is currently available on the SIMTEL20.ARPA Ada Software repository, in the directory PD:<PIWG.ADA>.

The performance evaluation tests for the prototype feature (enumeration representation clauses) and a sample of the tests for interrupt response time is given below. The machine readable form of these tests is available from the Air Force Institute of Technology, Department of Mathematics and Computer Science (ENC), WPAFB OH 45433.

A13 3 3C Source Code

This benchmark was designed as a prototype to learn more about benchmarking Chapter 13 features. As a test for the

efficiency of a compiler's implementation of enumeration representation clauses it is not complete because it tests only the successor ('SUCC) attribute. A complete benchmark would have to include tests for a wide range of enumeration objects with all of the enumeration type attributes and operations.

```
-----
-- Test Name:  A13_3_3C
-- Author:     Capt Dan Joyce
-- Date:       8 Jun 1987
-- Test Description: This test is designed to determine the
--                   processing overhead associated with enumeration types
--                   whose representations have been modified with an
--                   enumeration representation clause from Chapter 13
-----
```

with A000050; use A000050;

with REMOTE_GLOBAL ; use REMOTE_GLOBAL ; --control optimization

procedure A13_3_3C is -- main procedure to execute

```
type enum_type is (a00,a01,a02,a03,a04,a05,a06,a07,a08,a09,
                   a10,a11,a12,a13,a14,a15,a16,a17,a18,a19,
                   a20,a21,a22,a23,a24,a25,a26,a27,a28,a29,
                   a30,a31,a32,a33,a34,a35,a36,a37,a38,a39,
                   a40,a41,a42,a43,a44,a45,a46,a47,a48,a49,
                   a50,a51,a52,a53,a54,a55,a56,a57,a58,a59,
                   a60,a61,a62,a63,a64,a65,a66,a67,a68,a69,
                   a70,a71,a72,a73,a74,a75,a76,a77,a78,a79,
                   a80,a81,a82,a83,a84,a85,a86,a87,a88,a89,
                   a90,a91,a92,a93,a94,a95,a96,a97,a98,a99,
                   a100);
```

```
type enum_type2 is (a00,a01,a02,a03,a04,a05,a06,a07,a08,a09,
                   a10,a11,a12,a13,a14,a15,a16,a17,a18,a19,
                   a20,a21,a22,a23,a24,a25,a26,a27,a28,a29,
                   a30,a31,a32,a33,a34,a35,a36,a37,a38,a39,
                   a40,a41,a42,a43,a44,a45,a46,a47,a48,a49,
                   a50,a51,a52,a53,a54,a55,a56,a57,a58,a59,
                   a60,a61,a62,a63,a64,a65,a66,a67,a68,a69,
                   a70,a71,a72,a73,a74,a75,a76,a77,a78,a79,
                   a80,a81,a82,a83,a84,a85,a86,a87,a88,a89,
                   a90,a91,a92,a93,a94,a95,a96,a97,a98,a99,
                   a100);
```

```
for enum_type2 use (0,10,20,30,40,50,60,70,80,90,
                   100,110,120,130,140,150,160,170,180,190,
                   200,210,220,230,240,250,260,270,280,290,
                   300,310,320,330,340,350,360,370,380,390,
                   400,410,420,430,440,450,460,470,480,490,
```



```

500,510,520,530,540,550,560,570,580,590,
600,610,620,630,640,650,660,670,680,690,
700,710,720,730,740,750,760,770,780,790,
800,810,820,830,840,850,860,870,880,890,
900,910,920,930,940,950,960,970,980,990,
1000);

enum_object  : enum_type := a00;
enum_object2 : enum_type2 := a00;
--
begin
  A000052A;
  --
  -- Control loop
  --
  for J in 1 .. 10 loop
    GLOBAL := 0 ;
    enum_object := enum_type'first;
    for INSIDE_LOOP in 1 .. 100 loop
      GLOBAL := GLOBAL + A_ONE ; -- typical control loop is
      REMOTE ;                  -- these two statements
      enum_object := enum_type'succ(enum_object);
    end loop ;
  end loop ;

  A000053A;
  A000054A;
  --
  -- Test loop
  --
  for J in 1 .. 10 loop
    GLOBAL := 0 ;
    enum_object2 := enum_type2'first;
    for INSIDE_LOOP in 1 .. 100 loop
      GLOBAL := GLOBAL + A_ONE;
      REMOTE ;
      enum_object2 := enum_type2'succ(enum_object2);
    end loop ;
  end loop ;
  --
  a000055A;
end A13_3_3C;

```

A000050 Source Code

```

-- A000050
-- This is a package that contains modified versions of A000051
-- thru A000054. These procedures save the wall and cpu time
-- in variables in the A000050 package rather than writing them
-- to a file each time a measurement is taken. I am concerned
-- that the overhead for the I/O is distorting the timing
-- measurements. Therefore, the data will be written at the
-- end, after all measurements have been taken
--

```

```

-- USAGE :      A000052A      (1)
--              control procedure
--              A000053A      (2)
--              A000054A      (3)
--              test procedure
--              A000055A      (4)
-- RESULT :
--              ( (4) - (3) ) - ( (2) - (1) ) is the measurement
--
--              The second expression takes out the time to make the
--              measurement. As a check, (3) - (2) should be close to
--              (2) - (1)
with CPU_TIME_CLOCK ; -- various choices on tape
with CALENDAR ; -- used for WALL clock times
with TEXT_IO ; -- for printing times
with DURATION_IO ; -- for printing times

package A000050 is
CPU_SECONDS_START_CONTROL      : DURATION;
CPU_SECONDS_STOP_CONTROL      : DURATION;
CPU_SECONDS_START_TEST        : DURATION;
CPU_SECONDS_STOP_TEST         : DURATION;
CPU_SECONDS_DIFF_CONTROL      : DURATION;
CPU_SECONDS_DIFF_TEST         : DURATION;
CPU_SECONDS_DIFF              : DURATION;

WALL_SECONDS_START_CONTROL    : DURATION;
WALL_SECONDS_STOP_CONTROL    : DURATION;
WALL_SECONDS_START_TEST      : DURATION;
WALL_SECONDS_STOP_TEST       : DURATION;
WALL_SECONDS_DIFF_CONTROL    : DURATION;
WALL_SECONDS_DIFF_TEST       : DURATION;
WALL_SECONDS_DIFF            : DURATION;

procedure A000052A;
procedure A000053A;
procedure A000054A;
procedure A000055A;
end A000050;

package body A000050 is
--
procedure A000052A is

begin
    CPU_SECONDS_START_CONTROL := CPU_TIME_CLOCK;
    WALL_SECONDS_START_CONTROL :=
CALENDAR.SECONDS(CALENDAR.CLOCK);
end A000052A;
--

```

```

procedure A000053A is

begin
    CPU_SECONDS_STOP_CONTROL := CPU_TIME_CLOCK;
    WALL_SECONDS_STOP_CONTROL :=
    CALENDAR.SECONDS(CALENDAR.CLOCK);
end A000053A;
--

procedure A000054A is

begin
    CPU_SECONDS_START_TEST := CPU_TIME_CLOCK;
    WALL_SECONDS_START_TEST := CALENDAR.SECONDS(CALENDAR.CLOCK);
end A000054A;
--

procedure A000055A is

MY_FILE :TEXT_IO.FILE_TYPE;
begin
    CPU_SECONDS_STOP_TEST := CPU_TIME_CLOCK;
    WALL_SECONDS_STOP_TEST := CALENDAR.SECONDS(CALENDAR.CLOCK);
    --
    --
    CPU_SECONDS_DIFF_TEST := CPU_SECONDS_STOP_TEST -
                             CPU_SECONDS_START_TEST;
    CPU_SECONDS_DIFF_CONTROL := CPU_SECONDS_STOP_CONTROL -
                                CPU_SECONDS_START_CONTROL;
    CPU_SECONDS_DIFF := CPU_SECONDS_DIFF_TEST -
                        CPU_SECONDS_DIFF_CONTROL;

    WALL_SECONDS_DIFF_TEST := WALL_SECONDS_STOP_TEST -
                              WALL_SECONDS_START_TEST;
    WALL_SECONDS_DIFF_CONTROL := WALL_SECONDS_STOP_CONTROL -
                                 WALL_SECONDS_START_CONTROL;
    WALL_SECONDS_DIFF := WALL_SECONDS_DIFF_TEST -
                          WALL_SECONDS_DIFF_CONTROL;
    --
    TEXT_IO.CREATE(MY_FILE, TEXT_IO.OUT_FILE, 'A000050D');
    --
    TEXT_IO.NEW_LINE(MY_FILE);
    TEXT_IO.PUT(MY_FILE, ' CPU SECONDS DIFF CONTROL: ');
    DURATION_IO.PUT(MY_FILE,CPU_SECONDS_DIFF_CONTROL);
    TEXT_IO.NEW_LINE(MY_FILE,1);
    TEXT_IO.PUT(MY_FILE, ' CPU SECONDS DIFF TEST: ');
    DURATION_IO.PUT(MY_FILE,CPU_SECONDS_DIFF_TEST);
    TEXT_IO.NEW_LINE(MY_FILE,1);
    TEXT_IO.PUT(MY_FILE, ' CPU SECONDS DIFFERENCE: ');
    DURATION_IO.PUT(MY_FILE,CPU_SECONDS_DIFF);
    --
    TEXT_IO.NEW_LINE(MY_FILE,2);
    TEXT_IO.PUT(MY_FILE, 'WALL SECONDS DIFF CONTROL: ');
    DURATION_IO.PUT(MY_FILE,WALL_SECONDS_DIFF_CONTROL);
    TEXT_IO.NEW_LINE(MY_FILE,1);

```

```

TEXT_IO.PUT(MY_FILE, 'WALL SECONDS DIFF TEST:      ');
DURATION_IO.PUT(MY_FILE, WALL_SECONDS_DIFF_TEST);
TEXT_IO.NEW_LINE(MY_FILE, 1);
TEXT_IO.PUT(MY_FILE, 'WALL SECONDS DIFFERENCE:      ');
DURATION_IO.PUT(MY_FILE, WALL_SECONDS_DIFF);
TEXT_IO.CLOSE(MY_FILE);
end A000055A;
end A000050;

```

Sample Benchmark for Interrupt Delay Time

```

-----
-- INT_TEST4
--
--      This is a benchmark that will measure the interrupt delay
--      time associated with a task as an interrupt handler.
--      Task INT_HANDLE_2_TO_14 has an entry tied to interrupt 3
--      with an address clause. The accept block of this task
--      should be entered whenever interrupt 3 is raised.
--      This benchmark will raise MIL-STD-1750A interrupt 3
--      (a floating point overflow) by causing a floating point
--      overflow in the main procedure.
--      This benchmark calls RESET_INT_VECTORS_7_AND_9 and
--      GET_ALL_TIMES in package TIME_PACKAGE_1750A to set up and
--      retrieve precise CPU time measurements that are not
--      available with package CALENDAR.
--      This benchmark has 1 TASK with 10 entries, each tied to
--      a 1750a interrupt.
--      Author:      Capt Dan Joyce
--      Date: 27 Sept 87 1325
--      Version:      2.4
-----

```

```

package GLOBAL_INTERRUPT_MAKER4 is
  function FLOAT_RETURN return float;
end GLOBAL_INTERRUPT_MAKER4;

```

```

with TIME_PACKAGE_1750A; use TIME_PACKAGE_1750A;
with GLOBAL_INTERRUPT_MAKER4;
with TEXT_IO; use TEXT_IO;
procedure INT_TEST4 is

```

```

  package time_1750a_io is new FLOAT_IO(time_1750a);
  use time_1750a_io;

```

```

  float_object      : FLOAT              := 0.500000 * 2.0 ** 127;

```

```

  max_values : constant integer := 100;
  type TIME_ARRAY_TYPE is array (1..max_values) of TIME_1750A;

```

```

-----
-- Variables for statistical calculations
-- The address clause is used so those memory

```

```

-- locations may be examined on the 1750A to
-- verify the accuracy of the values reported
-----
mean      : TIME_1750A := 0.0;
for mean use at 16*6000*;

variance  : TIME_1750A := 0.0;
for variance use at 16*6004*;

sum_del_tim  : TIME_1750A := 0.0;
for sum_del_tim use at 16*6010*;

sum_del_tim2 : TIME_1750A := 0.0;
for sum_del_tim2 use at 16*6014*;

-----
-- Time collection Variables
-----
before_interrupt : TIME_ARRAY_TYPE := (others => 0.0);
after_interrupt  : TIME_ARRAY_TYPE := (others => 0.0);

clock_bias_start : TIME_ARRAY_TYPE := (others => 0.0);
clock_bias_stop  : TIME_ARRAY_TYPE := (others => 0.0);

int_delay        : TIME_ARRAY_TYPE := (others => 0.0);
for int_delay use at 16*7000*;

start_benchmark   : TIME_1750A := 0.0;
for start_benchmark use at 16*6020*;

stop_benchmark    : TIME_1750A := 0.0;
for stop_benchmark use at 16*6024*;

main_timera_return : TIME_1750A := 0.0;
main_dummyb_return  : TIME_1750A := 0.0;

task_timera_return  : TIME_1750A := 0.0;
pragma SHARED(task_timera_return);
task_dummya_return   : TIME_1750A := 0.0;
pragma SHARED(task_dummya_return);
task_dummyb_return   : TIME_1750A := 0.0;
pragma SHARED(task_dummyb_return);

-----
-- Interrupt Handling Task Specifications
-----
task INT_HANDLE_2_TO_14 is
  pragma PRIORITY(3);
  entry interrupt_2;
  for interrupt_2 use at 2;
  entry interrupt_3;
  for interrupt_3 use at 3;

```

```

entry interrupt_4;
for interrupt_4 use at 4;
entry interrupt_6;
for interrupt_6 use at 6;
entry interrupt_8;
for interrupt_8 use at 8;
entry interrupt_10;
for interrupt_10 use at 10;
entry interrupt_11;
for interrupt_11 use at 11;
entry interrupt_12;
for interrupt_12 use at 12;
entry interrupt_13;
for interrupt_13 use at 13;
entry interrupt_14;
for interrupt_14 use at 14;
end INT_HANDLE_2_TO_14;

```

```

-----
-- Interrupt Handling Task Bodies
-----

```

```

-----
-- This is the task that will handle the overflow --
-----

```

```

task body INT_HANDLE_2_TO_14 is

```

```

    begin

```

```

        loop

```

```

            select

```

```

                accept interrupt_2 do

```

```

                    get_all_times(task_dummya_return,
                                task_dummyb_return);

```

```

                    put_line('in interrupt_2 accept');

```

```

                end interrupt_2;

```

```

            or

```

```

                accept interrupt_3 do

```

```

                    -----
                    -- This is the accept that handles the --
                    -- floating point overflow                --
                    -----

```

```

                    get_all_times(task_timera_return,
                                task_dummyb_return);

```

```

                end interrupt_3;

```

```

            or

```

```

                accept interrupt_4 do

```

```

                    get_all_times(task_dummya_return,
                                task_dummyb_return);

```

```

                    put_line('in interrupt_4 accept');

```

```

                end interrupt_4;

```

```

            or

```

```

                accept interrupt_6 do

```

```

                    get_all_times(task_dummya_return,
                                task_dummyb_return);

```

```

        put_line('in interrupt_6 accept');
    end interrupt_6;
or
accept interrupt_8 do
    get_all_times(task_dummya_return,
                  task_dummyb_return);
    put_line('in interrupt_8 accept');
end interrupt_8;
or
accept interrupt_10 do
    get_all_times(task_dummya_return,
                  task_dummyb_return);
    put_line('in interrupt_10 accept');
end interrupt_10;
or
accept interrupt_11 do
    get_all_times(task_dummya_return,
                  task_dummyb_return);
    put_line('in interrupt_11 accept');
end interrupt_11;
or
accept interrupt_12 do
    get_all_times(task_dummya_return,
                  task_dummyb_return);
    put_line('in interrupt_12 accept');
end interrupt_12;
or
accept interrupt_13 do
    get_all_times(task_dummya_return,
                  task_dummyb_return);
    put_line('in interrupt_13 accept');
end interrupt_13;
or
accept interrupt_14 do
    get_all_times(task_dummya_return,
                  task_dummyb_return);
    put_line('in interrupt_14 accept');
end interrupt_14;
end select;
end loop;
end INT_HANDLE_2_TO_14;

```

```

-----
-- This procedure calculates the mean and variance
-- for the interrupt delay time
-----

```

```

procedure STATISTICS_AND_RESULTS is
    elapsed_bench_time : time_1750a := 0.0;
    n                  : time_1750a := 0.0;

begin
    mean           := 0.0;
    variance       := 0.0;
    sum_del_tim    := 0.0;

```

```

sum_del_tim2      := 0.0;

for i in 1 .. max_values loop
    -----
    -- The bias of the GET_ALL_TIMES call is
    -- added back in
    -----
    int_delay(i) := after_interrupt(i) -
                    (before_interrupt(i) + clock_bias_stop(i) -
                     clock_bias_start(i) );
    sum_del_tim := sum_del_tim + int_delay(i);
    sum_del_tim2 := sum_del_tim2 +
                    (int_delay(i) * int_delay(i));
end loop;

n := time_1750a(max_values);
mean := sum_del_tim / n;
variance := (sum_del_tim2 - (n*mean*mean)) / ( n - 1.0 );

new_line;
put('start :');
put(start_benchmark);
put(' stop : ');
put(stop_benchmark);
put(' elapsed time : ');
elapsed_bench_time := stop_benchmark - start_benchmark;
put(elapsed_bench_time);

new_line;
put(' n = ');
put(n);
put(' mean = ');
put(mean);
put(' var = ');
put(variance);
new_line;

end STATISTICS_AND_RESULTS;

begin -- INT_TEST4

    RESET_INT_VECTORS_7_AND_9;
    GET_ALL_TIMES(start_benchmark,
                  main_dummyb_return);

    for i in 1 .. max_values loop
        -----
        -- The first 2 clock calls are used to factor out
        -- the time a GET_ALL_TIMES call will take
        -----

```



```

    GET_ALL_TIMES(clock_bias_start(i),
                  main_dummyb_return);
    GET_ALL_TIMES(clock_bias_stop(i),
                  main_dummyb_return);
    GET_ALL_TIMES(before_interrupt(i),
                  main_dummyb_return);
    float_object :=
        float_object + GLOBAL_INTERRUPT_MAKER4.FLOAT_RETURN;

        -- This will cause an overflow

    after_interrupt(i) := task_timera_return;

end loop;

GET_ALL_TIMES(stop_benchmark,
              main_dummyb_return);

abort INT_HANDLE_2_TO_14;
STATISTICS_AND_RESULTS;

end INT_TEST4;

package body GLOBAL_INTERRUPT_MAKER4 is
    function FLOAT_RETURN return float is
    begin
        return 0.50000 * 2.0 ** 127;
    end FLOAT_RETURN;
end GLOBAL_INTERRUPT_MAKER4;

```

Appendix D: Validation Test Results

This appendix contains the results of the validation tests that were run on the three compilers in this thesis. A compiler either passes the test, fails the test, or a test is ruled inapplicable because the compiler does not support the feature in the manner it is tested, indicated by N/A.

Enumeration Representation Clauses

<u>Test</u>	<u>Compiler A</u>
BD3001A	Passed
BD3002A	Passed
BD3004A	Passed
BD3012A	Passed

Address Clauses and Interrupts

<u>Test</u>	<u>Compiler A</u>	<u>Compiler B</u>	<u>Compiler C</u>
BD5001A	Passed	Passed	Passed
BD5002A	Passed	Passed	Passed
BD5002B	N/A	N/A	N/A
BD5002C	N/A	N/A	N/A
BD5002D	N/A	N/A	N/A
BD5002E	N/A	N/A	N/A
AD5003A	Passed	Passed	Passed
AD5003B	Passed	Passed	Passed
AD5003C	Passed	Passed	Passed
AD5003D	Passed	Passed	Passed
AD5003E	Passed	Passed	Passed
BD5004A	Passed	Passed	Passed
BD5004B	Passed	Passed	Passed
BD5004C	Passed	Passed	Passed
BD5005A	Passed	Passed	Passed
BD5005B	Passed	Passed	Passed

<u>Test</u>	<u>Compiler A</u>	<u>Compiler B</u>	<u>Compiler C</u>
BD5006A	Passed	Passed	Passed
BD5006B	Passed	Passed	Passed
BD5006C	Passed	Passed	Passed
BD5006D	Passed	Passed	Passed
BD5006E	Passed	Passed	Passed
AD5007A	N/A	N/A	N/A
AD5007B	N/A	N/A	N/A
AD5007C	N/A	N/A	N/A
BD5007A	N/A	N/A	N/A
BD5008A	Passed	Passed	Passed
AD5011A	Passed	Passed	Passed
AD5011B	Passed	Passed	Passed
AD5011C	Passed	Passed	Passed
AD5011D	Passed	Passed	Passed
AD5011E	Passed	Passed	Passed
AD5011F	Passed	Passed	Passed
AD5011G	Passed	Passed	Passed
AD5011H	Passed	Passed	Passed
AD5011I	Passed	Passed	Passed
AD5011J	Passed	Passed	Passed
AD5011K	N/A	N/A	N/A
AD5011L	N/A	N/A	N/A
AD5011M	N/A	N/A	N/A
AD5011N	N/A	N/A	N/A
AD5011O	N/A	N/A	N/A
AD5011P	N/A	N/A	N/A
AD5011Q	N/A	N/A	N/A
AD5012A	Passed	Passed	Passed
AD5012B	Passed	Passed	Passed
AD5012C	Passed	Passed	Passed
AD5012D	Passed	Passed	Passed
AD5012E	Passed	Passed	Passed
AD5012F	Passed	Passed	Passed
AD5012G	Passed	Passed	Passed
AD5012H	Passed	Passed	Passed
AD5012I	Passed	Passed	Passed
AD5012J	Passed	Passed	Passed
AD5012K	N/A	N/A	N/A
AD5012L	N/A	N/A	N/A
AD5012M	N/A	N/A	N/A

<u>Test</u>	<u>Compiler A</u>	<u>Compiler B</u>	<u>Compiler C</u>
AD5012N	N/A	N/A	N/A
AD5012O	N/A	N/A	N/A
AD5012P	N/A	N/A	N/A
AD5012Q	N/A	N/A	N/A
AD5013A	Passed	Passed	Passed
AD5013B	Passed	Passed	Passed
AD5013C	Passed	Passed	Passed
AD5013D	Passed	Passed	Passed
AD5013E	Passed	Passed	Passed
AD5013F	Passed	Passed	Passed
AD5013G	Passed	Passed	Passed
AD5013H	Passed	Passed	Passed
AD5013I	Passed	Passed	Passed
AD5013J	Passed	Passed	Passed
AD5013K	N/A	N/A	N/A
AD5013L	N/A	N/A	N/A
AD5013M	N/A	N/A	N/A
AD5013N	N/A	N/A	N/A
AD5013O	N/A	N/A	N/A
AD5013P	N/A	N/A	N/A
AD5013Q	N/A	N/A	N/A
AD5013R	N/A	N/A	N/A
AD5013S	N/A	N/A	N/A
AD5014A	Passed	Passed	Passed
AD5014B	Passed	Passed	Passed
AD5014C	Passed	Passed	Passed
AD5014D	Passed	Passed	Passed
AD5014E	Passed	Passed	Passed
AD5014F	Passed	Passed	Passed
AD5014G	Passed	Passed	Passed
AD5014H	Passed	Passed	Passed
AD5014I	Passed	Passed	Passed
AD5014J	Passed	Passed	Passed
AD5014K	N/A	N/A	N/A
AD5014L	N/A	N/A	N/A
AD5014M	N/A	N/A	N/A
AD5014N	N/A	N/A	N/A
AD5014O	N/A	N/A	N/A
AD5014P	N/A	N/A	N/A
AD5014Q	N/A	N/A	N/A
AD5014R	N/A	N/A	N/A
AD5014S	N/A	N/A	N/A

<u>Test</u>	<u>Compiler A</u>	<u>Compiler B</u>	<u>Compiler C</u>
AD5021A	N/A	N/A	N/A
AD5022A	N/A	N/A	N/A
AD5031A	N/A	N/A	N/A
AD5032A	N/A	N/A	N/A
AD5041A	N/A	N/A	N/A
AD5042A	N/A	N/A	N/A
BD5101A	N/A	Passed	N/A
BD5101B	N/A	Passed	N/A
BD5102A	N/A	FAILED ⁵	N/A
BD5103A	N/A	Passed	N/A
BD5104A	N/A	Passed	N/A
AD5111A	N/A	Passed	N/A

<u>Totals</u>	<u>Compiler A</u>	<u>Compiler B</u>	<u>Compiler C</u>
Passed	58	63	58
Failed	0	1	0
N/A	52	46	52

⁵ This failed test was passed by a subsequent version of compiler B.

Appendix E: Performance Evaluation Test Results

Results of Prototype Benchmark

The results shown below are the differences in the control and test loop execution times, in seconds, of the A13_3_3C benchmark (test_loop_time - control_loop_time); they are presented here as an example of the results obtained from this benchmark. The numbers in square brackets index the trial number of the first result in that row.

Chapter 4 describes how the test version of the benchmark could be faster than the control version, generating the negative difference.

Compiler A Results

	-0.0300	-0.0100	0.0200	0.0700	0.0400
	0.0001	0.0099	0.0	-0.0601	-0.0100
[11]	0.0100	0.0200	-0.0200	0.0	-0.0100
	0.0	-0.0001	-0.0200	0.0100	0.0200
[21]	0.0200	-0.0100	0.0200	0.0100	0.0990
	-0.4800	-0.4600	-0.4700	-0.4401	-0.4200
[31]	0.0100	0.0	-0.0300	-0.0100	0.0
	0.0200	0.0001	-0.0200	0.0199	-0.0200
[41]	-0.0300	0.0	-0.0200	0.0	0.0990
	0.0	0.0199	0.0100	0.0990	0.0
[51]	-0.0099	-0.0700	-0.0100	0.0100	0.0300
	-0.0400	0.0200	-0.0200	0.0100	-0.0100
[61]	0.0400	0.0099	0.0200	0.0	-0.0100
	-0.0200	0.0200	0.0	-0.0100	0.0100
[71]	-0.0100	0.0	0.0100	-0.0200	0.0100

Mean = -0.0264453 Seconds,
Standard Deviation = 0.1186431 Seconds.

Appendix F: Detailed Description of Time Package 1750A

Introduction

This appendix describes the detailed development of the TIME_PACKAGE_1750A, which contains three procedures used to return the current elapsed time from the timer registers available on the MIL-STD-1750A architecture. This appendix explains how each of these procedures was designed, followed by the assembly language and Ada source code for these procedures. Much of the information found in this appendix is extracted from MIL-STD-1750A (DoD, 1982) -- the reader is directed to that document for further detail on interrupt handling and assembly instructions for the MIL-STD-1750A architecture.

A crucial link in the development of benchmarks to measure interrupt response time was the development of a method for precise, accurate measurement of the elapsed CPU time for this operation. TIME_PACKAGE_1750A consists of three functions: (1) an assembly language routine that resets and redefines the interrupt service routines for MIL-STD-1750A interrupts, (2) an assembly routine to retrieve the current value of the two MIL-STD-1750A timer registers and the number of interrupts that have occurred, and (3) an Ada package that other Ada procedures can 'with' to retrieve CPU times. Each of these will be described in detail below.

Background

Among its general purpose registers, the MIL-STD-1750A has two 16-bit registers known as Timer A and Timer B. When a MIL-STD-1750A processor is started up, or reset, both Timer A and Timer B are set to zero and are incremented every 10 and 100 microseconds, respectively, and count in the following sequence:

0000₁₆, 0001₁₆, ..., 7FFF₁₆, 8000₁₆, ..., FFFF₁₆, 0000₁₆,
0001₁₆ ...

Whenever the timers increment from FFFF₁₆ to 0000₁₆, they each generate a MIL-STD-1750A interrupt. Timer A generates interrupt 7 and Timer B generates interrupt 9 (DoD, 1982:19). The current elapsed time, then, may be calculated using the following formulas:

$$El_a = [(I_a * 65,536) + T_a] * 10 / 1000000.0 \text{ seconds}$$

or,

$$El_b = [(I_b * 65,536) + T_b] * 100 / 1000000.0 \text{ seconds}$$

where

El_a = Elapsed time (using Timer A)

El_b = Elapsed time (using Timer B)

I_a = number of Timer A interrupts

I_b = number of Timer B interrupts

T_a = current value of Timer A register

T_b = current value of Timer B register

Although both Timers should report similar results, with Timer A having more precision, both timers were used in TIME_PACKAGE_1750A as a check to ensure that the values being

returned were valid. In order to calculate the elapsed time, TIME_PACKAGE_1750A must keep track of the number of times interrupts 7 and 9 have occurred, retrieve the values of the Timer A and B registers, and perform the conversion shown above.

Interrupt Counting

In order to count the number of interrupts 7 and 9 that occurred, I had to modify the interrupt linkage pointers for those two interrupts. I wrote an assembly language routine that is called RESETIV, which resets the service pointers for those interrupts.

Whenever an interrupt occurs, the current state of the CPU is saved, and the CPU reads a series of three 16-bit words starting at the memory location indicated by the value in the Service Pointer. The first two words are loaded into the Interrupt Mask and the Status Word, respectively. The third word contains the new instruction counter, and is the first instruction of the interrupt service routine. When RESETIV is called, it resets the service pointer for interrupt 7 (or 9) to another three words of memory, the third being the address of the first instruction of the new interrupt 7 (or 9) service routine declared in RESETIV. All this routine does is add one to memory location 500 (or 501 for interrupt 9). These memory locations are reserved for this use because two 16 bit integers are declared in the TIME_PACKAGE_1750A and assigned this address with an address clause.

RESETIV is linked to the Ada procedure
 RESET_INT_VECTORS_7_AND_9 with a *pragma* INTERFACE statement.
 When a procedure calls RESET_INT_VECTORS_7_AND_9 then the
 service pointers are reset and the interrupt service routines
 defined in RESETIV start keeping the interrupt count.

The RESETIV source code is given below.

RESETIV Source Code

```
*****
*
*
* RESETIV
*
* Name:          Reset Interrupt Vectors
* Description:   This assembly language routine will reset the
*               interrupt vectors for MIL-STD-1750A interrupts 7 and 9,
*               which correspond to the timer A and timer B clocks
*               resetting to 0000 (from FFFF), or 'wrapping around'. The
*               Interrupt service routines provided in this program will
*               increment a counter that keeps track of the number of
*               interrupts. By returning the current value of the timers
*               and the number of times they have wrapped around, one may
*               determine the current elapsed time without calling a
*               predefined [slower] system time routine, such as Ada's
*               (tm) CALENDAR package.
*
* References:
*   MIL-STD-1750A, (notice 1), 21 May 1982, U.S.
*   Printing Office. [Pages 19-21 explain interrupt pointers,
*   service routines, etc.]
*
* Author: Capt Dan Joyce (with much help from 1Lt Marc Pitarys)
* Date:   6 Sep 87
* Version: 1.4
*
*****
      EXPORT RESETIV
      MODULE RESETIV
;
R0      EQU      0
R1      EQU      1
R2      EQU      2
R3      EQU      3
R4      EQU      4
R5      EQU      5
```

```

R6      EQU      6
R7      EQU      7
R8      EQU      8
R9      EQU      9
R10     EQU      10
R11     EQU      11
R12     EQU      12
R13     EQU      13
R14     EQU      14
R15     EQU      15
;
;
JC       7,RESETIV      ; BRANCH TO THE PROGRAM BEGIN
ORG      00500
TICKNTA DATAT 0000      ;TickntA & B count the number of times
TICKNTB DATAT 0000      ;TimerA & B have wrapped around (0000)
;
;
IVEC7    DATAT      V7ISR
EVEN
IVEC9    DATAT      V9ISR
EVEN
V7ISR    DATAT      0          ;New Int Mask: MASK ALL INTERRUPTS
        DATAT      0          ;New Status Word
        DATAT      I7ISR      ;New Instruction Counter
EVEN
V9ISR    DATAT      0          ;New Int Mask: MASK ALL INTERRUPTS
        DATAT      0          ;New Status Word
        DATAT      I9ISR      ;New Instruction Counter
EVEN
;-----
;
;      INTERRUPT HANDLER FOR TIMER A
;
; IQW    REST      100
; EVEN
I7ISR    TAH                ; Stop Timer A
        TBH                ; Stop Timer B
        INCM      1,0500    ; INCREMENT THE CLOCK TICK COUNT
        TAS                ; Restart Timer A
        TBS                ; Restart Timer B
        ENBL          ; RE-ENABLE INTERRUPTS
        LSTI      0002E    ;Reload old status
;      [2E + 0] -> MK,
;      [2E + 1] -> SW,
;      [2E + 2] -> IC (RETURN)
;-----
;
;      INTERRUPT HANDLER FOR TIMER B
; EVEN
I9ISR    TAH                ; Stop Timer A
        TBH                ; Stop Timer B
        INCM      1,0501    ; INCREMENT THE CLOCK TICK COUNT
        TAS                ; Restart Timer A
        TBS                ; Restart Timer B

```

```

        ENBL          ; RE-ENABLE INTERRUPTS
        LSTI          00032 ; Reload old status (see above)
;-----
RESETIV DSBL          ; Disable Interrupts
        PSHM          R0,R1 ; Save R0 and R1
        SR            R0,R0 ; Set R0 = 0
        SMK           R0    ; Clear Interrupt Mask
        CLIR          ; CLEAR ANY PENDING INTERRUPTS.
        TAH           ; Stop Timer A and B and Reset them
        TBH           ; Both to Zero
        OTA           R0
        OTB           R0

        L             R1,IVEC7 ;SET UP INTERRUPT VECTORS FOR TIMER A
        ST            R1,0002F ;02F is Int Svc Ptr Addr for Int 7

        L             R1,IVEC9 ; SET UP INTERRUPT VECTORS FOR TIMERB
        ST            R1,00033 ; 033 is Int Svc Ptr Addr for Int 9

        RCFR          R0      ;CLEAR THE FAULT REGISTER
        CLIR          ; Clear Pending Interrupts
        ENBL          ; Enable Interrupts Any NEW ONES
        LIM           R0,05940; Reset the interrupt mask

        SMK           R0      ;
        POPM          R0,R1   ; Restore R0 and R1
        URS           R15
        END

```

Returning the Interrupt Count and Timer Register Contents

The assembly language routine GET_TIMERS is called from the procedure GET_ALL_TIMES in package TIME_PACKAGE_1750A and passes two long_integer (32-bit) objects as arguments. When the GET_TIMERS routine is entered, registers 3 and 4 contain the addresses of the two values that the GET_ALL_TIMES procedure is expecting back. GET_TIMERS retrieves the interrupt count for interrupt 7/9 from memory location 500/501 and stores this in the lower half of the 32 bit return variable. This has the same effect as multiplying the interrupt count by 65,536. GET_TIMERS then uses a

MIL-STD-1750A XIO instruction to retrieve the values of the timer registers and stores these values in the upper half of the return variables indicated by the addresses in registers 3 and 4.

Registers 3 and 4 were used in the assembly routine after the TIME_PACKAGE_1750A was initially compiled and I found that registers 3 and 4 were being used by the Ada procedure to pass the return addresses of the output parameters to the assembly routine. When writing assembly language routines that will be called from Ada, one must first determine the registers the Ada calling routine is using to pass parameters to the called assembly language routine. GET_TIMERS is linked to an Ada procedure, also called GET_TIMERS, using the pragma interface.

The assembly language source code for GET_TIMERS is given below.

GET TIMERS Source Code

```
*****
* Name:      GET_TIMERS
* Author:    Capt Dan Joyce
* Date:      31 Aug 87
* Version:   1.2
* Description: This is a 1750A assembly language routine
*              designed to return the values of the timer A and Timer B
*              clocks to a predefined location in memory. This routine
*              must be used with the TIME_PACKAGE. The timers will be
*              loaded in memory locations identified in register 3 and 4:
*              TimerA => [R3,R3+1]
*              TimerB => [R4,R4+1]
*              The value of the timer is loaded at the upper half of the
*              double word (higher memory) The lower half will contain
*              the number of interrupts that have occurred, so the entire
*              32 bit word will contain the total number of ticks. For
*              example, assume that 3 interrupts have taken place on
*              timer A, and the current value of the timer A register is
```

```

*      25 (hex). The value for [R3,R3+1] will be (in hex)
*      00030025.
*      The number of interrupts will be updated by modified
*      Interrupt service routines for interrupt 7 (Timer A) and
*      interrupt 9 (Timer B) in the assembly routine RESETIV,
*      which is tied to the Ada procedure
*      RESET_INT_VECTORS_7_AND_9 with a pragma interface in
*      TIME_PACKAGE_1750A. The number of interrupts must be
*      stored at location 500 (timer A) and 501 (hex) for Timer
*      B. See the Source for RESETIV.ASM.

```

```

*      This must be called using pragma interface:

```

```

*      Ada equivalent:  procedure get_timers(timer_a_ret,
*                                     timer_b_ret);

```

```

*****

```

```

MODULE      GET_TIMERS
EXPORT      GET_TIMERS

;
R0          EQU      0
R1          EQU      1
R2          EQU      2
R3          EQU      3
R4          EQU      4
R12         EQU      12
R14         EQU      14
R15         EQU      15
;
GET_TIMERS  EQU      *

      DSBL                      ; Disable interrupts so clock
                                ; Can't wrap around during
                                ; routine
      PSHM      R0,R14          ; Save Registers R0,R1
      XORR      R0,R0          ; Clear R0

      L         R2,0500         ; Load # Timer A intpts
      STBX      R12,R3          ; Store contents of R2 at
                                ; [R3+0] (R12-12 = R0)

      L         R2,0501         ; Load # Timer B intpts
      STBX      R12,R4          ; Store contents of R2 at
                                ; [R4 + 0] (R12 - 12 = R0)

      AIM       R3,1            ; Add 1 to Reg 3 and R4 so they
      AIM       R4,1            ; point to the next locations
                                ; in memory (second 16 bits)

      ITA       R2              ; R2 <- Timer A
      STBX      R12,R3          ; Store contents of R2 at
                                ; [R3 + 0 ] (R12 - 12 = R0)

      ITB       R2              ; R2 <- Timer B

```

```

      STBX      R12,R4      ; Store contents of R2 at
                          ; [R4 + R0] (R12 - 12 = R0)

      POPM      R0,R14     ; Restore Registers R0 to R14
      ENBL      ; Allow Interrupts again
      URS       R15        ; return
      END

```

Packaging the Procedures for Use

TIME_PACKAGE_1750A, finally, groups all of these procedures in one place and defines an interface through the entirely Ada procedure GET_ALL_TIMES, which converts the integral values of Timer A and B "ticks" to a floating point value in seconds. Floating point was used instead of fixed point because the possibility of losing precision is greater with fixed point numbers, even using deltas of 0.00001 and 0.0001. The comments in the source code explain how to use the package.

TIME PACKAGE 1750A Source Code

```

-----
-- TIME_PACKAGE_1750A
--
-- This package contains functions and procedures that will
-- return elapsed time to the caller.
--
-- INSTRUCTIONS FOR USING THIS PACKAGE:
--
-- 1) 'With' the package (and 'use' the package if you
-- don't want to make qualified calls).
--
-- 2) Declare at least two objects of type TIME_1750A
-- (defined in this package) as the parameters returned by the
-- procedure GET_ALL_TIMES, eg.
--      timera_return : TIME_1750A := 0.0;
--      timerb_return : TIME_1750A := 0.0;
--
-- 3) Make a call to RESET_INT_VECTORS_7_AND_9. This should
-- be the first executable statement in the program using this

```

```

-- package. RESET_INT_VECTORS_7_AND_9 allows
-- TIME_PACKAGE_1750A to calculate the elapsed time.
-- 4) Make calls to GET_ALL_TIMES, eg:
--      GET_ALL_TIMES(timer_a_return,timer_b_return);
--
-- Author:      Capt Dan Joyce
-- Date:        06 Sep 1987
-- Version:     1.6
-----
with SYSTEM;
package TIME_PACKAGE_1750A is

    subtype time_1750A is LONG_FLOAT range 0.0 .. 1.0E30;

    current_total_aticks : long_integer := 0;
    current_total_bticks : long_integer := 0;

    -----
    -- These two memory locations are used by the interrupt
    -- Service routines as storage for the interrupt count
    -- Get_Timers will read the number of timer a/b interrupts
    -- from these locations. The Address Clause keeps Ada from
    -- using these locations for anything else.
    -----

    timera_interrupt_count : integer := 0;
    timerb_interrupt_count : integer := 0;
    for timera_interrupt_count use at 16#0500#;
    for timerb_interrupt_count use at 16#0501#;

    procedure RESET_INT_VECTORS_7_AND_9;
    pragma interface(assembly,reset_int_vectors_7_and_9,
                    'resetiv');

    procedure GET_TIMERS(aticks_return : out long_integer;
                        bticks_return : out long_integer);
    pragma interface(assembly,get_timers,'get_timers');

    procedure GET_ALL_TIMES( elapsed_timera : out time_1750A;
                            elapsed_timerb : out time_1750A);

end TIME_PACKAGE_1750A;

package body TIME_PACKAGE_1750A is

    procedure GET_ALL_TIMES ( elapsed_timera : out time_1750A;
                            elapsed_timerb : out time_1750A) is

        begin

            GET_TIMERS(current_total_aticks,
                      current_total_bticks);

```



```

if current_total_aticks < 0 then
    current_total_aticks := 0;
end if;

if current_total_bticks < 0 then
    current_total_bticks := 0;
end if;

elapsed_timera :=
    TIME_1750A( current_total_aticks) / 100_000.0;

elapsed_timerb :=
    TIME_1750A( current_total_bticks) / 10_000.0;

end GET_ALL_TIMES;

end TIME_PACKAGE_1750A;

```

Appendix G: Two-Sample t Test Calculations

This appendix contains the two-sample t test calculations used to test the hypothesis that the mean interrupt delay time for INT_TEST1 was less than that for INT_TEST2, INT_TEST3, or INT_TEST4; and the hypothesis that the mean interrupt delay for INT_TEST5 was less than that for INT_TEST6.

All hypothesis testing was done at the 0.01 level of significance. The sample size for all tests was 100, thus the critical value for the one-sided test was

$$t_{0.01,198} = 2.33 \text{ (Larsen and Marx, 1986:580).}$$

The formula for the test statistic for the two sample t test is:

$$t_{xy} = (\bar{x} - \bar{y}) / [s_p (1/n + 1/m)^{1/2}] \quad (1)$$

where

s_p = pooled variance,

n = sample size of the X population,

m = sample size of the Y population.

The subscript on variables in this appendix refers to the interrupt benchmark number, i.e. \bar{x}_2 is the mean for INT_TEST2 and s_{p12} is the pooled variance for INT_TEST1 and INT_TEST2. The following pooled variances were calculated from the sample variances reported in the benchmarks:

$$s_{p12} = 7.234 * 10^{-6} \quad s_{p14} = 6.721 * 10^{-6}$$

$$s_{p13} = 7.567 * 10^{-6} \quad s_{p56} = 7.009 * 10^{-6}$$

The following sets of hypotheses were tested:

$$\begin{array}{ll} H_{o12}: \mu_1 \geq \mu_2 & \text{vs.} \quad H_{a12}: \mu_1 < \mu_2 \\ H_{o13}: \mu_1 \geq \mu_3 & \text{vs.} \quad H_{a13}: \mu_1 < \mu_3 \\ H_{o14}: \mu_1 \geq \mu_4 & \text{vs.} \quad H_{a14}: \mu_1 < \mu_4 \\ H_{o56}: \mu_5 \geq \mu_6 & \text{vs.} \quad H_{a56}: \mu_5 < \mu_6 \end{array}$$

Using formula 1 for the test statistic and the sample means reported in Table IV in Chapter 5, the following test statistics were calculated:

$$\begin{array}{ll} t_{12} = -35.776 & t_{14} = -56.918 \\ t_{13} = -55.414 & t_{56} = -19.168 \end{array}$$

In all cases, these values are less than -2.33, the critical t value, therefore all H_o hypotheses are rejected.

Bibliography

- Ada Joint Program Office. Ada Compiler Validation Procedures and Guidelines. Washington: AJPO, 1 January 1987.
- Ada Information Clearinghouse. Validated Ada Compilers List. Washington: AJPO, 1 September 1987.
- Altman, Neal. "Factors Causing Unexpected Variations in Ada Benchmarks," Draft Report, May 1987, Software Engineering Institute, Pittsburgh, PA. (Report number SEI-87-MR-12)
- Bassman, Mitchell J. and others. "An Approach for Evaluating the Performance Efficiency of Ada Compilers," Ada Letters, 5: 151-163 (Sept, Oct 1985).
- Booch, Grady. Software Engineering with Ada, Second Ed. Menlo Park California: The Benjamin/Cummings Publishing Company, 1987.
- Boeing Military Airplane Company (BMAC), "Ada Compiler Evaluation Capability Operational Software: Software Requirements Specification." Document Number S500-11703. BMAC, Wichita KS, 5 August 1987.
- Bennett, SSgt. James, Personal Interviews. ASD/ENASF, Wright-Patterson AFB OH, June-July 1987.
- Benwell, Nicholas, ed. Benchmarking: Computer Evaluation and Measurement. Washington, D.C.: Hemisphere Publishing Corporation, 1975.
- Brashear, Philip, ACVC Maintenance Manager. Personal Interviews. SofTech, Inc., Dayton OH, July 1987a.
- , ACVC Maintenance Manager. Personal Correspondence. SofTech, Inc., Dayton OH, 30 September 1987b.
- Bunce, Philip. Handouts distributed at "Ada/MIL-STD-1750A Issues" Tutorial. Ada-JOVIAL User's Group Meeting, Dayton OH, 13 July 1987.
- Chitwood, Georgeanne, Ada Validation Facility Manager. Personal Interview. ASD/SCOL, Wright-Patterson AFB OH, April 1987.
- Clapp, Russell M. and others. "Toward Real-Time Performance Benchmarks for Ada," Communications of the ACM, 29: 760-778 (Aug 1986).
- Clements, Paul, Software Engineering Applications Section Chief. Telephone Interviews. Naval Research Lab, Washington, D.C., July 1987.

Conn, Richard. The Ada Software Repository and the Defense Data Network. New York: New York Zeotrope, 1987.

Craine, David B. Ada Compiler Evaluation Techniques for Real-Time Avionics Applications. MS Thesis, AFIT/GCS/MA/86D-6. School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, December 1986.

Department of Defense. Military Standard: Sixteen-Bit Computer Instruction Set Architecture. MIL-STD-1750A. Washington: U.S. Government Printing Office, 21 May 1982.

Department of Defense. Military Standard: Ada Programming Language. ANSI/MIL-STD-1815A. Washington: Department of Defense, 22 January 1983.

Department of Defense. Military Standard: JOVIAL (J73). MIL-STD-1589C. Washington: Department of Defense, 6 July 1984.

Department of Defense. Use of Ada in Weapon Systems. DoD Directive 3405.2. Washington: Government Printing Office, 30 March 1987.

Goodenough, John B. The Ada Compiler Validation Capability Implementers' Guide: Version 1. SofTech, Inc., Waltham, MA, December 1986.

Johnson, Conrad, Senior Software Engineer. Telephone Interview. Soncraft, Inc., Chicago IL, 10 September 1987.

King, Capt, David, Software Group Lead. Personal Interview. ASD/ENASF, Wright-Patterson AFB OH, March 1987.

Klemens, John D. Examination of the Effects of Using Ada in Flight Control Software. MS Thesis, AFIT/GCS/MA/87D-3. School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, December 1987.

Larsen, Richard J. and Morris L. Marx. An Introduction to Mathematical Statistics and Its Applications. Englewood Cliffs, NJ: Prentice-Hall, 1986.

Lyons, Maj, Robert. 'Ada Insertion into ATA, ATF, and LHX - A Tri-Service Perspective', presentation to Ada-JOVIAL User's Group, Dayton OH, 14 July 1987.

Myers, Ware. 'Ada: First Users - Pleased; Prospective Users - Still Hesitant,' Computer, 20:71 (March 1987).

Performance Issues Working Group (PIWG). Ada Slices: Official Newsletter of ACM SIGAda PIWG. December 1986.

Phillips, Stephen P. and Peter R. Stevenson. "The Role of Ada in Real Time Embedded Applications," Ada Letters, 6: 54-60 (Nov, Dec 1986).

Pitarys, 1Lt, Marc, Avionics Systems Engineer. Personal Interviews. AFWAL/AAAF-3, Wright-Patterson AFB, OH, July, 1987.

Ploedereder, Erhard. "Ada Compiler Validation," Application of Ada Higher Order Language to Guidance and Control, Paper 7, 1-8. NATO Advisory Group for Aerospace Research and Development. June, 1986 (AD-A171299).

Roark, Chuck and Ron McAfee. "The Applicability of Ada to MIL-STD-1750A," Unpublished article, Texas Instruments, Plano, TX, July 1987.

Roark, Chuck, Senior Member Technical Staff. Personal Interview. Texas Instruments, Dayton OH, 15 July 1987.

Seward, Dave, Principal Engineer. Telephone Interview. Advanced Computer Techniques, New York, New York, 15 July 1987.

Squire, Jon, Chairman, ACM SIGAda PIWG, Telephone Interviews. Westinghouse Defense and Electronics Center, Baltimore MD, April, June 1987.

Wilson, Steven, Ada Task Leader. Personal Interview. ASD/SCOL, Wright-Patterson AFB OH, April 1987a.

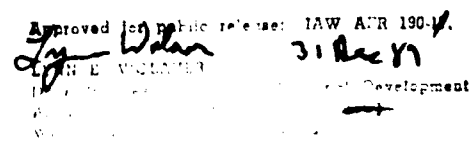
----- Using the ACVC Tests (Version 1.9). Unpublished Document. ASD/SCOL, Wright-Patterson AFB OH, April 1987b.

Witt, Donald J. Using Ada in the Real-Time Avionics Environment: Issues and Conclusions. MS Thesis, AFIT/GCS/MA/85D-6. School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, December 1985.

VITA

Captain Daniel O. Joyce was born on 21 October 1959 in Riverside, California. He graduated from Foxcroft Academy in Dover-Foxcroft, Maine in 1977 and attended the University of New Hampshire, from which he received the degree of Bachelor of Science in Mathematics in May 1981. Upon graduation, he received a commission in the USAF through the ROTC program. He completed technical training at Keesler AFB, Mississippi in January 1982 and was assigned as an Attrition Modeling Systems Analyst to 1851st Information Systems Support Squadron, Offutt AFB, Nebraska. He entered the School of Engineering, Air Force Institute of Technology, in June of 1986.

Permanent address: c/o John A. Glover
Box 265
Monson, Maine 04464

REPORT DOCUMENTATION PAGE				Form Approved OMB No. 0704-0188	
1a REPORT SECURITY CLASSIFICATION Unclassified			1b RESTRICTIVE MARKINGS		
2a SECURITY CLASSIFICATION AUTHORITY			3 DISTRIBUTION / AVAILABILITY OF REPORT Approved for public release; distribution unlimited.		
2b DECLASSIFICATION / DOWNGRADING SCHEDULE					
4 PERFORMING ORGANIZATION REPORT NUMBER(S) AFIT/GCS/MA/87D-2			5 MONITORING ORGANIZATION REPORT NUMBER(S)		
6a NAME OF PERFORMING ORGANIZATION School of Engineering		6b OFFICE SYMBOL (if applicable) AFIT/ENG	7a NAME OF MONITORING ORGANIZATION		
6c ADDRESS (City, State, and ZIP Code) Air Force Institute of Technology (AU) Wright-Patterson AFB, Ohio 45433-6583			7b ADDRESS (City, State, and ZIP Code)		
8a NAME OF FUNDING / SPONSORING ORGANIZATION Systems Engineering Avionics Facility		8b OFFICE SYMBOL (if applicable) ASD/ENASF	9 PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER		
8c ADDRESS (City, State, and ZIP Code) ASD/ENASF Wright-Patterson AFB, Ohio 45433			10 SOURCE OF FUNDING NUMBERS		
			PROGRAM ELEMENT NO	PROJECT NO	TASK NO
11 TITLE (Include Security Classification) VALIDATING AND EVALUATING ADA'S REPRESENTATION CLAUSES AND IMPLEMENTATION-DEPENDENT FEATURES ON MIL-STD-1750A ARCHITECTURE					
12 PERSONAL AUTHOR(S) Daniel O. Joyce, Capt, USAF					
13a TYPE OF REPORT MS Thesis		13b TIME COVERED FROM _____ TO _____		14 DATE OF REPORT (Year, Month, Day) 1987 December	
15 PAGE COUNT 132					
16 SUPPLEMENTARY NOTATION					
17 COSATI CODES			18 SUBJECT TERMS (Continue on reverse if necessary and identify by block number)		
FIELD	GROUP	SUB-GROUP	Ada Compilers Benchmark Validation; Embedded computer		
12	05				
12	08				
19 ABSTRACT (Continue on reverse if necessary and identify by block number)					
Thesis Advisor: Richard R. Gross, Lt Col, USAF Assistant Dean, School of Engineering					
					
20 DISTRIBUTION / AVAILABILITY OF ABSTRACT <input type="checkbox"/> UNCLASSIFIED/UNLIMITED <input checked="" type="checkbox"/> SAME AS RPT <input type="checkbox"/> DTIC USERS			21 ABSTRACT SECURITY CLASSIFICATION Unclassified		
22a NAME OF RESPONSIBLE INDIVIDUAL Richard R. Gross, Lt Col, USAF			22b TELEPHONE (Include Area Code) (513) 255-4372		22c OFFICE SYMBOL AFIT/EN

Abstract

Developers of applications for embedded systems need full implementations for all of the representation clauses and implementation-dependent features in Chapter 13 of the 'Language Reference Manual' (LRM) if they are to be successful in developing these application entirely in Ada. Because implementations of Ada's representation clauses and implementation-dependent features vary from compiler to compiler, these features must be validated and evaluated before they are used in applications that have such high reliability requirements. This thesis describes an approach used to develop validation tests and performance evaluation tests, or benchmarks, for Ada's address clauses and interrupts features and reports the results of the validation tests and benchmarks.

The validation tests were compiled with three validated Ada compilers, two of which were targeted to the MIL-STD-1750A processor. The benchmarks developed in this research measure interrupt delay time for interrupts associated with a task entry by an address clause. These benchmarks were compiled with a validated Ada compiler targeted to the MIL-STD-1750A and run on a Sperry 1631 MIL-STD-1750A processor.

EMD
DATE
FILMED
3-1988
DTIC